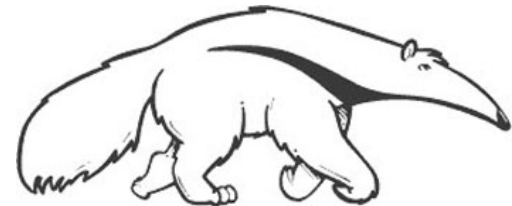# Introduction to Artificial Intelligence

CS171, Fall Quarter, 2019
Introduction to Artificial Intelligence
Prof. Richard Lathrop

Read Beforehand: All assigned reading so far

Midterm Review

- Agents: R&N Chap 2.1-2.3

- State Space Search: R&N Chap 3.1-3.7

- Local Search: R&N Chap 4.1-4.2

- Adversarial (Game) Search: R&N Chap 5.1-5.4

- Constraint Satisfaction: R&N Chap 6.1-6.4 (except 6.3.3)

- Propositional Logic: R&N Chap 7.1-7.5

Review Agents
Chapter 2.1-2.3

- Agent definition (2.1)

- Rational Agent definition (2.2)
  - Performance measure

- Task evironment definition (2.3)
  - PEAS acronym
  - Properties of task environments

Agents

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators

- Human agent:
  - Sensors: eyes, ears, …
  - Actuators: hands, legs, mouth…

- Robotic agent
  - Sensors: cameras, range finders, …
  - Actuators: motors

Agents and environments
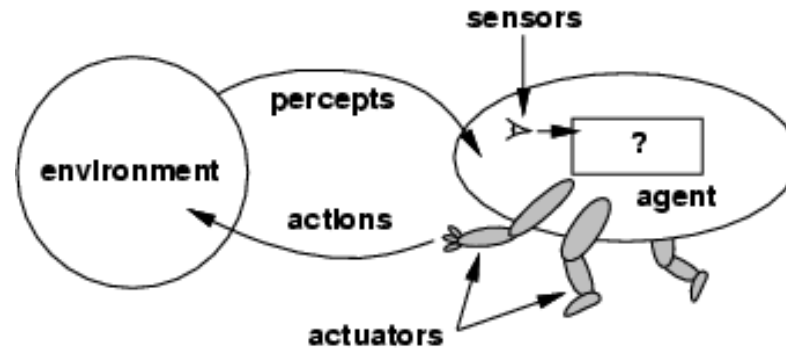
- Percept: agent's perceptual inputs at an instant
- The agent function maps from percept sequences to actions: $[f: \mathcal{P}^\star \rightarrow \mathcal{A}]$
- The agent program runs on the physical architecture to produce f
- agent = architecture + program

Rational agents

- Rational Agent: For each possible percept sequence, a rational agent should select an action that is *expected* to maximize its performance measure, based on the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

- Performance measure: An objective criterion for success of an agent's behavior   ("cost", "reward", "utility")

- E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

Task Environment

- Before we design an intelligent agent, we must specify its "task environment":

PEAS:

Performance measure
Environment
Actuators
Sensors

Environment types

- **Fully observable (vs. partially observable):** An agent's sensors give it access to the complete state of the environment at each point in time.

- **Deterministic (vs. stochastic):** The next state of the environment is completely determined by the current state and the action executed by the agent. (If the environment is deterministic except for the actions of other agents, then the environment is **strategic**.)

- **Episodic (vs. sequential):** An agent's action is divided into atomic episodes. Decisions do not depend on previous decisions/actions.

- **Known (vs. unknown):** An environment is considered to be "known" if the agent understands the laws that govern the environment's behavior.

Environment types

- **Static (vs. dynamic):** The environment is unchanged while an agent is deliberating. (The environment is **semidynamic** if the environment itself does not change with the passage of time but the agent's performance score does)

- **Discrete (vs. continuous):** A limited number of distinct, clearly defined percepts and actions.
  - How do we **represent** or **abstract** or **model** the world?

- **Single agent (vs. multi-agent):** An agent operating by itself in an environment. Does the other agent interfere with my performance measure?

Review State Space Search Chapter 3

- Problem Formulation (3.1, 3.3)
- Blind (Uninformed) Search (3.4)
  - Depth-First, Breadth-First, Iterative Deepening, Uniform-Cost, Bidirectional (if applicable)
  - Time? Space? Complete? Optimal?
- Heuristic Search (3.5)
  - A*, Greedy-Best-First

State-Space Problem Formulation

A **problem** is defined by five items:

(1) initial state e.g., "at Arad"

(2) actions *Actions(s)* = set of actions avail. in state s

(3) transition model Results(s,a) = state that results from action a in state s
   Alt: successor function *S(x)* = set of action–state pairs
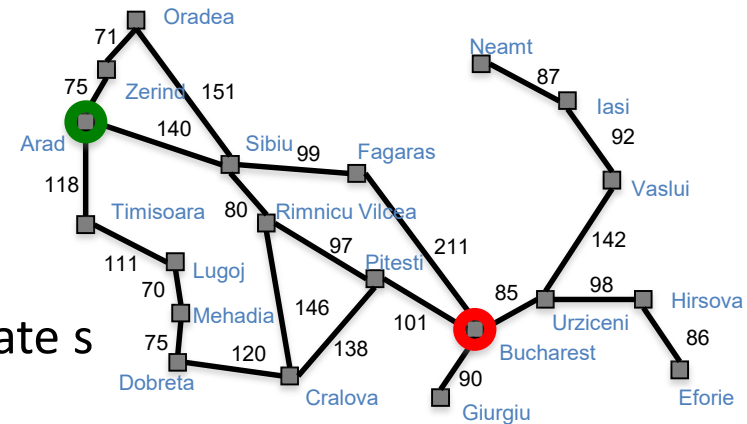   – e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*

(4) goal test, (or goal state)
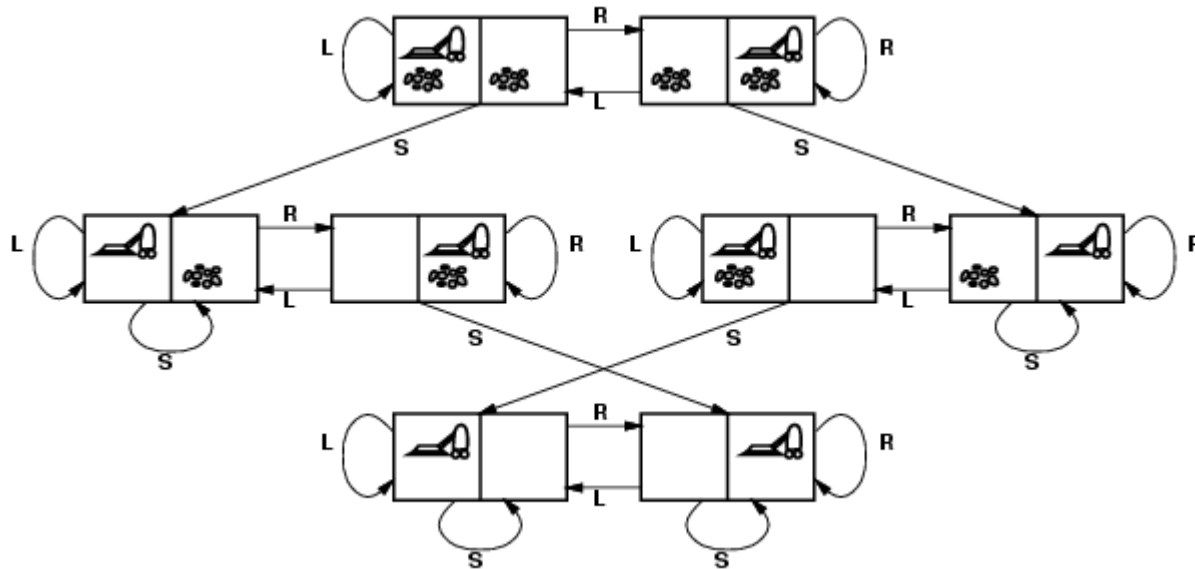e.g., *x* = "at Bucharest", *Checkmate(x)*

(5) path cost (additive)
   – e.g., sum of distances, number of actions executed, etc.
   – *c(x,a,y)* is the step cost, assumed to be ≥ 0 (and often, assumed to be $\geq \varepsilon > 0$)

A **solution** is a sequence of actions leading from the initial state to a goal state

Vacuum world state space graph

- **states?** discrete: dirt and robot locations
- **initial state?** any
- **actions?** *Left, Right, Suck*
- **transition model?** as shown on graph
- **goal test?** no dirt at all locations
- **path cost?** 1 per action

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration

- A node is a data structure constituting part of a search tree
- A node contains info such as:
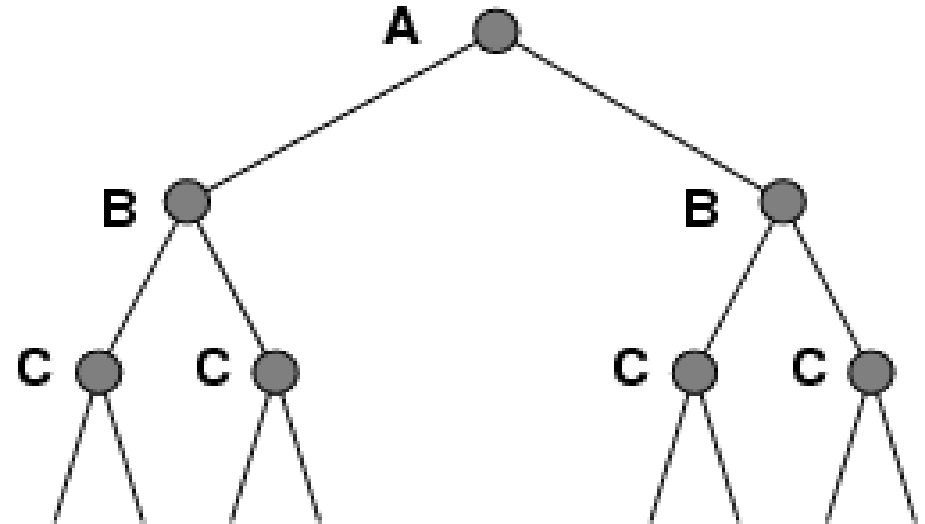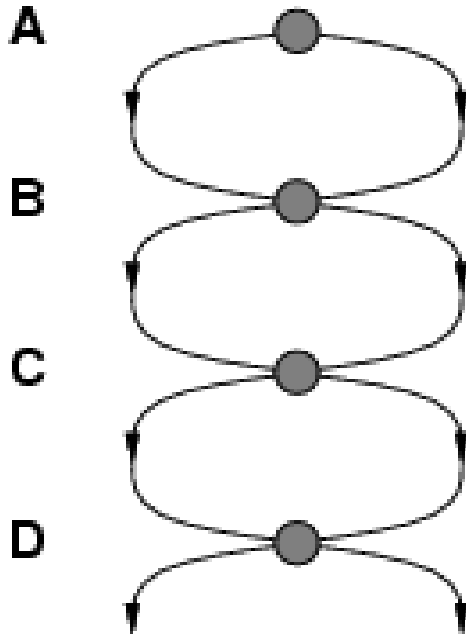 - state, parent node, action, path cost $g(x)$, depth, etc.
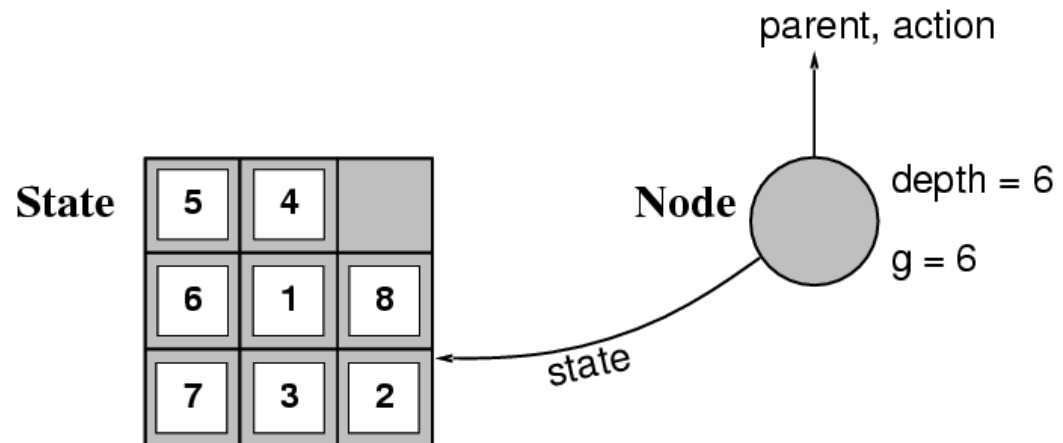
- The `Expand` function creates new nodes, filling in the various fields using the `Actions(S)` and `Result(S,A)` functions associated with the problem.

Tree search vs. Graph search Review Fig. 3.7, p. 77

- Failure to detect repeated states can turn a linear problem into an exponential one!

- Test is often implemented as a hash table.

Tree search vs. Graph search Review Fig. 3.7, p. 77

- What R&N call Tree Search vs. Graph Search
  - (And we follow R&N **exactly** in this class)
  - Has **NOTHING** to do with searching trees vs. graphs
- **Tree Search** = do **NOT** remember visited nodes
  - Exponentially slower search, but memory efficient
- **Graph Search** = **DO** remember visited nodes
  - Exponentially faster search, but memory blow-up
- **CLASSIC Comp Sci TIME-SPACE TRADE-OFF**

Solutions to Repeated States
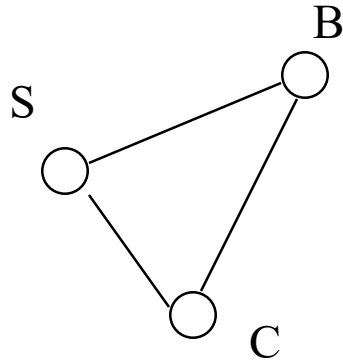
State Space
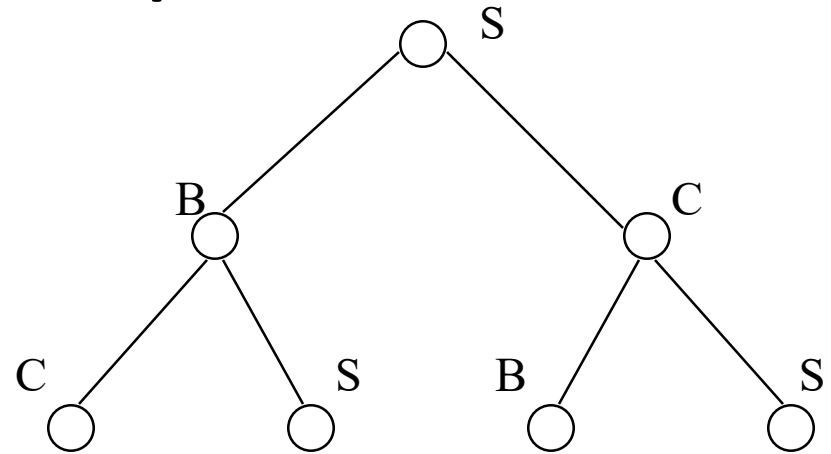
Example of a Search Tree

- Graph search ←——————— faster, but memory inefficient

  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid infinite loops by checking path back to root.

  - "visited?" test usually implemented as a <u>hash table</u>

Checking for identical nodes (1)
Check if a node is already in fringe-frontier

- It is "easy" to check if a node is already in the fringe/frontier (recall fringe = frontier = open = queue)
  - Keep a hash table holding all fringe/frontier nodes
    - Hash size is same O(.) as priority queue, so hash does not increase overall space O(.)
    - Hash time is O(1), so hash does not increase overall time O(.)
  - When a node is expanded, remove it from hash table (it is no longer in the fringe/frontier)
  - For each resulting child of the expanded node:
    - If child is not in hash table, add it to queue (fringe) and hash table
    - Else if an old lower- or equal-cost node is in hash, discard the new higher- or equal-cost child
    - Else remove and discard the old higher-cost node from queue and hash, and add the new lower-cost child to queue and hash

Always do this for tree or graph search in BFS, UCS, GBFS, and A*

Checking for identical nodes (2)
Check if a node is in explored/expanded

- It is memory-intensive [$O(b^d)$ or $O(b^m)$]to check if a node is in explored/expanded (recall explored = expanded = closed)
  - Keep a hash table holding all explored/expanded nodes (hash table may be HUGE!!)

- When a node is expanded, add it to hash (explored)

- For each resulting child of the expanded node:
  - If child is not in hash table or in fringe/frontier, then add it to the queue (fringe/frontier) and process normally (BFS normal processing differs from UCS normal processing, but the ideas behind checking a node for being in explored/expanded are the same).
  - Else discard any redundant node.

Always do this for graph search

Breadth-first graph search (R&N Fig. 3.11)

function BREADTH-FIRST-SEARCH(problem) **returns** a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0 **if** problem.GOAL-TEST(node.STATE) **then return** SOLUTION(node) frontier ← a FIFO queue with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) **then return** failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action **in** problem.ACTIONS(node.STATE) **do**
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier **then**
            if problem.GOAL-TEST(child.STATE) **then return** SOLUTION(child)
            frontier ← INSERT(child, frontier)

Figure 3.11   Breadth-first search on a graph.

Goal test before push

Avoid redundant frontier nodes

These three statements change tree search to graph search.

Properties of breadth-first search

- Complete? Yes, it always reaches a goal (if b is finite)
- Time? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

    (this is the number of nodes we generate)
- Space? $O(b^d)$

    (keeps every node in memory, either in frontier or on a path to frontier).
- Optimal?   No, for general cost functions.

        Yes, if cost is a non-decreasing function only of depth.
  - With $f(d) \geq f(d-1)$, e.g., step-cost = constant:
    - All optimal goal nodes occur on the same level
    - Optimal goals are always shallower than non-optimal goals
    - An optimal goal will be found before any non-optimal goal

- Usually Space is the bigger problem (more than time)

Uniform cost search (R&N Fig. 3.14) [A* is identical except queue sort = f(n)]

function UNIFORM-COST-SEARCH(problem) **returns** a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
   if EMPTY?(frontier) **then return** failure
   node ← POP(frontier) /* chooses the lowest-cost node in frontier */
   if problem.GOAL-TEST(node.STATE) **then return** SOLUTION(node)
   add node.STATE to explored
   for each action **in** problem.ACTIONS(node.STATE) **do**
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier **then**
         frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST **then**
         replace that frontier node with child

Goal test after pop

Avoid redundant frontier nodes

Avoid higher-cost frontier nodes

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

These three statements change tree search to graph search.

Uniform-cost search

Implementation: *Frontier* = queue ordered by path cost.
Equivalent to breadth-first if all step costs all equal.

- Complete? Yes, if b is finite and step cost ≥ ε > 0.
    (otherwise it can get stuck in infinite regression)

- Time? # of nodes with path cost ≤ cost of optimal solution.
    $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$

- Space? # of nodes with path cost ≤ cost of optimal solution.
    $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$.

- Optimal? Yes, for step cost ≥ ε > 0.

Depth-limited search & IDS (R&N Fig. 3.17-18)

function DEPTH-LIMITED-SEARCH(*problem, limit*) **returns** soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[*problem*]), *problem, limit*)

function RECURSIVE-DLS(*node, problem, limit*) **returns** soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
    else if DEPTH[*node*] = *limit* **then return** *cutoff*
    else for each *successor* **in** EXPAND(*node, problem*) **do**
        result ← RECURSIVE-DLS(*successor, problem, limit*)
        if *result* = *cutoff* **then** *cutoff-occurred?* ← true
        else if *result* ≠ *failure* **then return** *result*
    if *cutoff-occurred?* **then return** *cutoff* **else return** *failure*

Goal test in recursive call, one-at-a-time

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
    inputs: *problem*, a problem
    for *depth* ← 0 **to** ∞ **do**
        result ← DEPTH-LIMITED-SEARCH(*problem, depth*)
        if *result* ≠ cutoff **then return** *result*

At *depth* = 0, IDS only goal-tests the start node. The start node is is not expanded at *depth* = 0.

# Properties of iterative deepening search

- Complete?   Yes

- Time? $O(b^d)$

- Space? *O(bd)*

- Optimal?   No, for general cost functions.
        Yes, if cost is a non-decreasing function only of depth.

Generally the preferred uninformed search strategy.

Depth-First Search (R&N Section 3.4.3)

- Your textbook is ambiguous about DFS.
  - The second paragraph of R&N 3.4.3 states that DFS is an instance of Fig. 3.7 using a LIFO queue. Search behavior may differ depending on how the LIFO queue is implemented (as separate pushes, or one concatenation).
  - The third paragraph of R&N 3.4.3 says that an alternative implementation of DFS is a recursive algorithm that calls itself on each of its children, as in the Depth-Limited Search of Fig. 3.17 (above).
- **<u>For quizzes and exams, we will follow Fig. 3.17.</u>**
  - Generally, for tests DFS will be used only as an example.

Properties of depth-first search
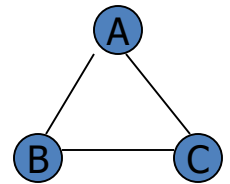
- Complete? No: fails in loops/infinite-depth spaces
  - Can modify to avoid loops/repeated states along path
    - check if current nodes occurred before on path to root
  - Can use graph search (remember all nodes ever seen)
    - problem with graph search: space is exponential, not linear
  - Still fails in infinite-depth spaces (may miss goal entirely)

- Time? $O(b^m)$ with m =maximum depth of space
  - Terrible if m is much larger than d
  - If solutions are dense, may be much faster than BFS

- Space? $O(bm)$, i.e., linear space!
  - Remember a single path + expanded unexplored nodes

- Optimal? No: It may find a non-optimal goal first

Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both "meet in the middle"
  - need to keep track of the intersection of 2 open sets of nodes

- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?

- Complexity
  - time complexity is best: $O(2 \, b^{(d/2)}) = O(b^{(d/2)})$
  - memory complexity is the same as time complexity
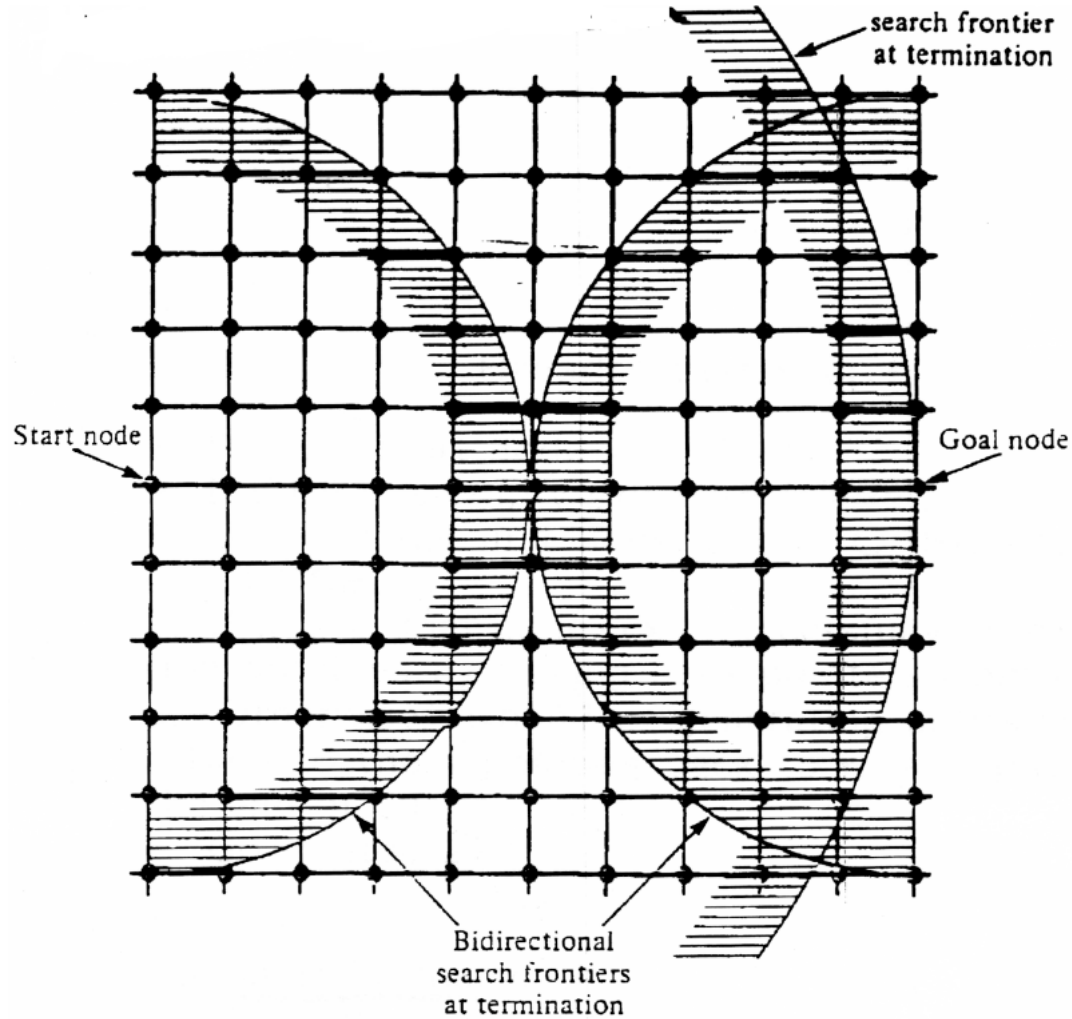
Bi-Directional Search

Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

Blind Search Strategies (3.4)

- Depth-first: Add successors to front of queue

- Breadth-first: Add successors to back of queue

- Uniform-cost: Sort queue by path cost $g(n)$

- Depth-limited: Depth-first, cut off at limit l

- Iterated-deepening: Depth-limited, increasing l

- Bidirectional: Breadth-first from goal, too.

- **Review "Example hand-simulated search"**
  - Lecture on "Uninformed Search"

Search strategy evaluation

- A search **strategy** is defined by **the order of node expansion**

- Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)
  - (UCS: **C*:** true cost to optimal goal; **ε > 0:** minimum step cost)

Summary of algorithms
Fig. 3.21, p. 91

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

There are a number of footnotes, caveats, and assumptions.
See Fig. 3.21, p. 91.
[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
    (also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
    (also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Generally the preferred
uninformed search strategy

Summary

- Generate the search space by applying actions to the initial state and all further resulting states.

- Problem: initial state, actions, transition model, goal test, step/path cost

- Solution: sequence of actions to goal

- Tree-search (don't remember visited nodes) vs. Graph-search (do remember them)

- Search strategy evaluation: b, d, m (UCS: C*, ε)
  - Complete? Time? Space? Optimal?

Heuristic function (3.5)

- Heuristic:
  - Definition: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  - "using rules of thumb to find answers"

- Heuristic function h(n)
  - Estimate of (optimal) cost from n to goal
  - Defined using only the *state* of node *n*
  - h(n) = 0 if n is a goal node
  - Example: straight line distance from n to Bucharest
    - Note that this is not the true state-space distance
    - It is an estimate – actual state-space distance can be higher

 - Provides problem-specific knowledge to the search algorithm

Relationship of search algorithms

- Notation:
  - *g(n)* = known cost so far to reach *n*
  - *h(n)* = estimated optimal cost from *n* to goal
  - *h*(n)* = true optimal cost from *n* to goal (unknown to agent)
  - *f(n) = g(n)+h(n)* = estimated optimal total cost through *n*

- Uniform cost search: sort frontier by *g(n)*
- Greedy best-first search: sort frontier by *h(n)*
- A* search: sort frontier by *f(n) = g(n) + h(n)*
  - Optimal for admissible / consistent heuristics
  - Generally the preferred heuristic search framework
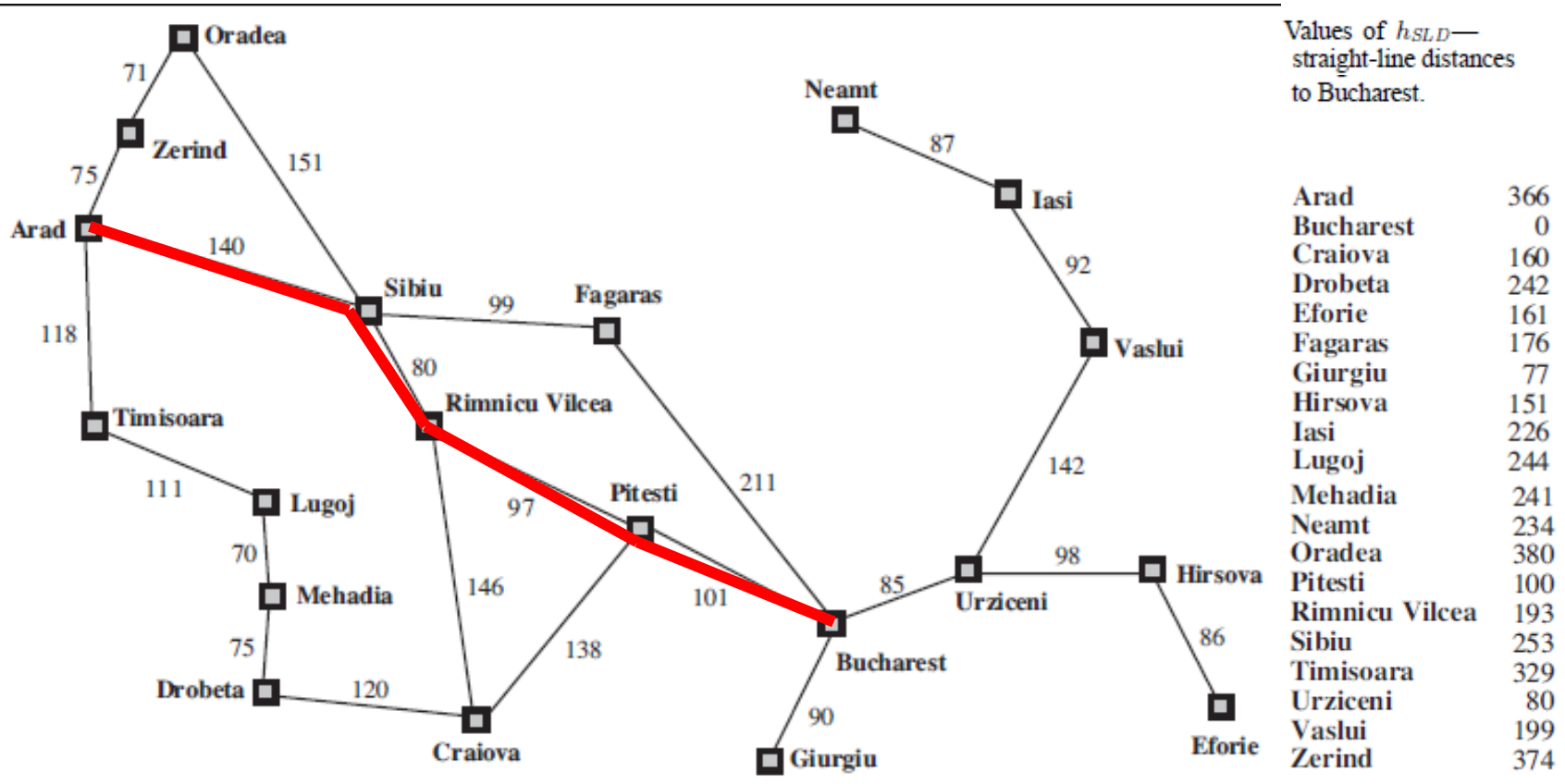 - Memory-efficient versions of A* are available: RBFS, SMA*

Greedy best-first search

- *h(n)* = estimate of cost from *n* to *goal*
  - e.g., *h(n)* = straight-line distance from *n* to Bucharest

- Greedy best-first search expands the node that appears to be closest to goal.
  - Sort queue by *h(n)*

- Not an optimal search strategy
  - May perform well in practice

Greedy best-first search example
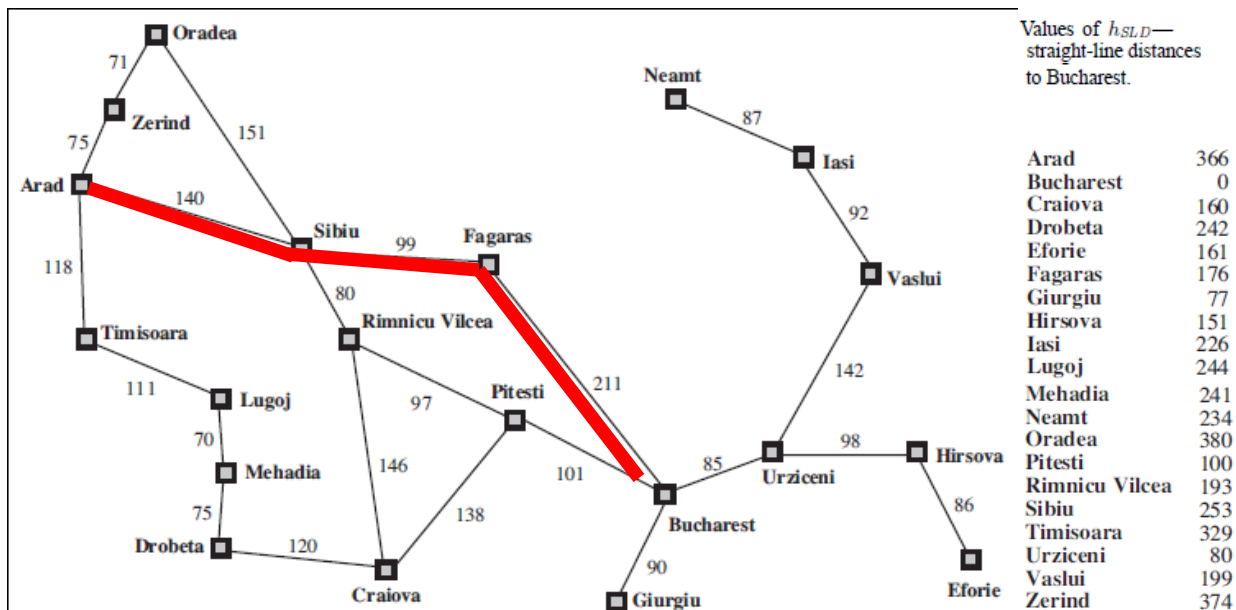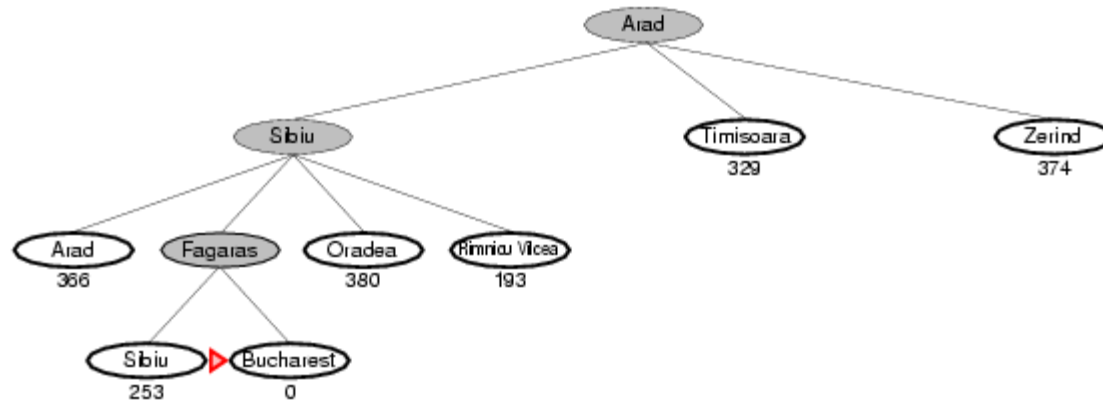
# Optimal Path

Properties of greedy best-first search

- <u>Complete?</u>
  - Tree version can get stuck in loops.
  - Graph version is complete in finite spaces.
- <u>Time?</u> *O(b^m)*
  - A good heuristic can give **<u>dramatic</u>** improvement
- <u>Space?</u> *O(b^m)*
  - Graph search keeps all nodes in memory
  - A good heuristic can give **<u>dramatic</u>** improvement
- <u>Optimal?</u> No
  - E.g., Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest is shorter!

A* search

- **Idea: avoid paths that are already expensive**
  - Generally the preferred simple heuristic search
  - Optimal if heuristic is:

    admissible (tree search)/consistent (graph search)

- Evaluation function *f(n) = g(n) + h(n)*
  - g(n) = known path cost so far to node n.
  - h(n) = <u>estimate</u> of (optimal) cost to goal from node n.
  - f(n) = g(n)+h(n)

    = <u>estimate</u> of total cost to goal through node n.

- *Priority queue sort function = f(n)*

A* tree search example: Simulated queue. City/f=g+h
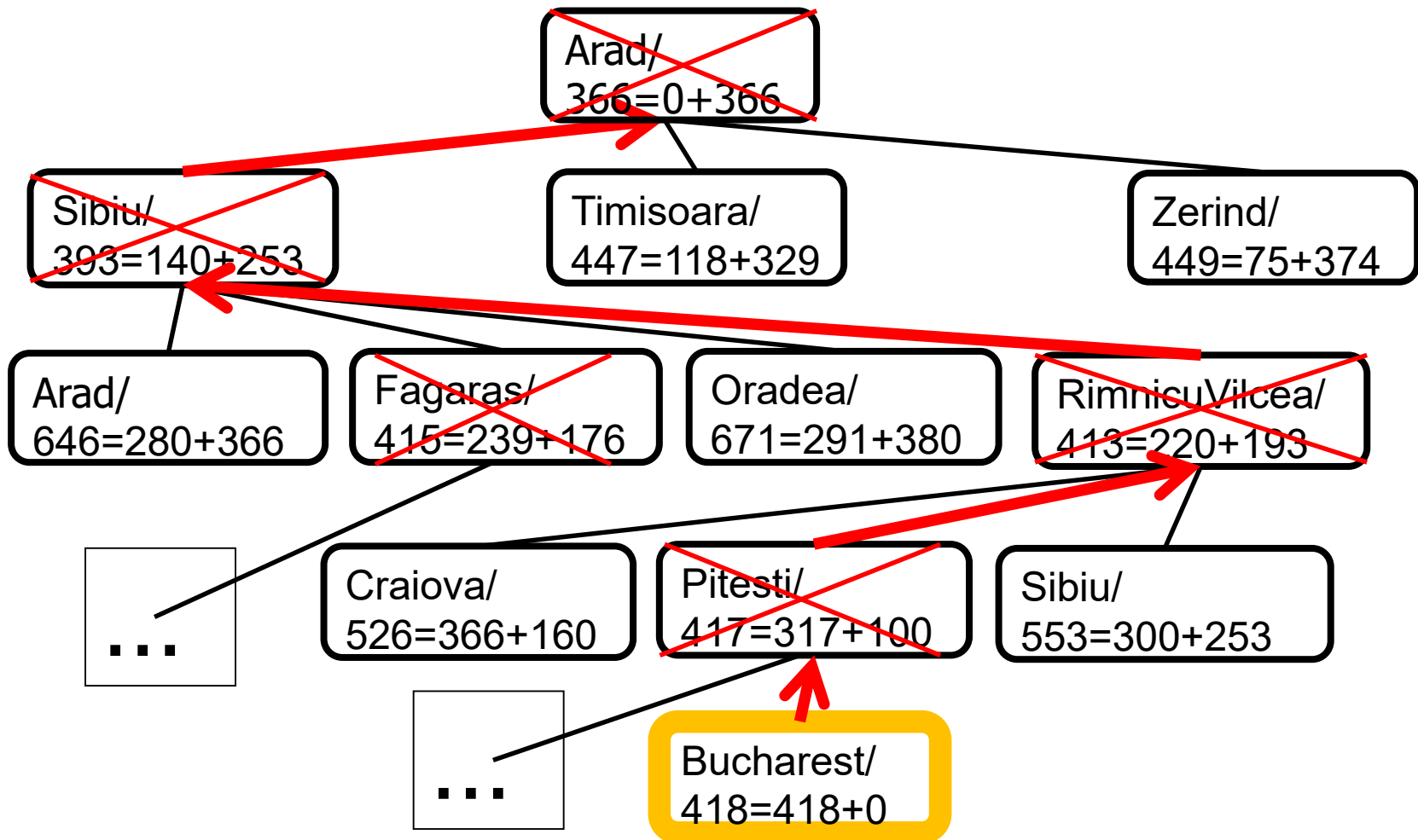
Properties of A*

- Complete? Yes

  (unless there are infinitely many nodes with *f ≤ f(G)*;

  can't happen if step-cost ≥ ε > 0)

- Time/Space? Exponential *O(b^d)*

  except if: $|h(n) - h^*(n)| \leq O(\log h^*(n))$

- Optimal?

  (with: Tree-Search, admissible heuristic;

  Graph-Search, consistent heuristic)

- *Optimally Efficient?*

  (no optimal algorithm with same heuristic is guaranteed to expand fewer nodes)

Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n,

  $h(n) \leq h^*(n),$ where $h^*(n)$ is the true cost to reach the goal state from n.

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

- Theorem: If $h(n)$ is admissible, A^* using `TREE-SEARCH` is optimal

Consistent heuristics (consistent => admissible)

- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,

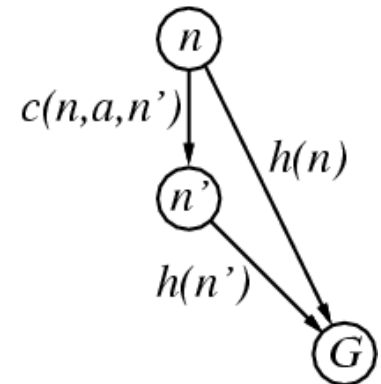$$h(n) \leq c(n,a,n') + h(n')$$

- If h is consistent, we have

$f(n') = g(n') + h(n')$ (by def.)

     $= g(n) + c(n,a,n') + h(n')$ $(g(n')=g(n)+c(n.a.n'))$

     $\geq g(n) + h(n) = f(n)$ (consistency)

$f(n')$ $\geq f(n)$

It's the triangle inequality !

- i.e., $f(n)$ is non-decreasing along any path.

- Theorem:
  If $h(n)$ is consistent, A* using `GRAPH-SEARCH` is optimal

keeps all checked nodes in memory to avoid repeated states

Optimality of A* (proof)
Tree Search, where *h(n)* is admissible

- Suppose some suboptimal goal G_2 has been generated and is in the frontier. Let *n* be an unexpanded node in the frontier such that *n* is on a shortest path to an optimal goal *G*.

> **We want to prove:**
> $$f(n) < f(G2)$$
> **(then A* will expand n before G2)**

- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $f(G) = g(G)$ since $h(G) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal

- $f(G_2) > f(G)$ from above, with h=0
- $h(n) \le h^*(n)$ since h is admissible (*under*-estimate)
- $g(n) + h(n) \le g(n) + h^*(n)$ from above
- $f(n) \le f(G)$ since g(n)+h(n)=f(n) & g(n)+h*(n)=f(G)
- $f(n) < f(G2)$ from above

Start

n

G

G_2
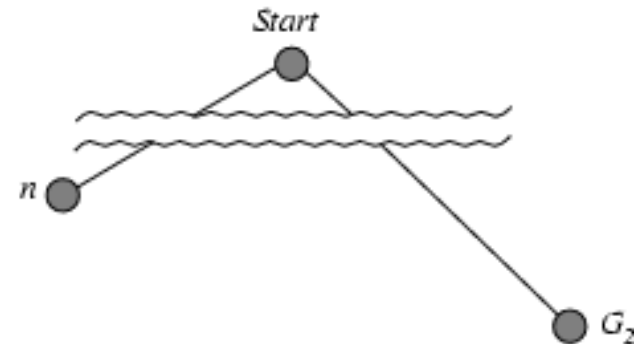
> R&N pp. 95-98 proves the optimality of A* graph search with a consistent heuristic

Dominance

- IF $h_2(n) \geq h_1(n)$ for all n
  THEN h_2 <u>dominates</u> h_1
  - h_2 is <u>almost always better</u> for search than h_1
  - h_2 <u>guarantees</u> to expand no more nodes than does h_1
  - h_2 <u>almost always</u> expands fewer nodes than does h_1
  - Not useful unless both h_1 & h_2 are admissible/consistent

- Typical 8-puzzle search costs
  (average number of nodes expanded):
  - *d=12*   IDS = 3,644,035 nodes
    $A^*(h_1)$ = 227 nodes
    $A^*(h_2)$ = 73 nodes
  - *d=24*   IDS = too many nodes
    $A^*(h_1)$ = 39,135 nodes
    $A^*(h_2)$ = 1,641 nodes

Review Local Search
Chapter 4.1-4.2, 4.6; Optional 4.3-4.5

- Problem Formulation (4.1)

- Hill-climbing Search (4.1.1)

- Simulated annealing search (4.1.2)

- Local beam search (4.1.3)

- Genetic algorithms (4.1.4)

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
  - Local search: widely used for *very big* problems
  - Returns good but *not optimal* solutions
  - *Usually very slow,* but can yield good solutions if you wait

- State space = set of "complete" configurations
- Find a complete configuration satisfying constraints
  - Examples: n-Queens, VLSI layout, airline flight schedules

- Local search algorithms
  - Keep a single "current" state, or small set of states
  - Iteratively try to improve it / them
  - Very memory efficient
    - keeps only one or a few states
    - You control how much memory you use

Random restart wrapper

- We'll use stochastic local search methods
  - Return different solution for each trial & initial state

- Almost every trial hits difficulties (see sequel)
  - Most trials will not yield a good result (sad!)

- Using many random restarts improves your chances
  - Many "shots at goal" may finally get a good one

- Restart a random initial state, *many times*
  - Report the best result found across *many* trials

Random restart wrapper

best_found ← **RandomState**()   // initialize to something

// now do repeated local search
loop do
  if (tired of doing it)
    then return *best_found*
  else
    result ← LocalSearch(**RandomState**())
    if (**Cost**(*result*) < **Cost**(*best_found*))
      // keep best result found so far
   **then** *best_found* ← *result*

> **You, as algorithm designer, write the functions named in red.**

Typically, **"tired of doing it"** means that some resource limit has been exceeded, e.g., number of iterations, wall clock time, CPU time, etc. It may also mean that result improvements are small and infrequent, e.g., less than 0.1% result improvement in the last week of run time.

Tabu search wrapper

- Add recently visited states to a tabu-list
  - Temporarily excluded from being visited again
  - Forces solver away from explored regions
  - Less likely to get stuck in local minima (hope, in principle)

- Implemented as a hash table + FIFO queue
  - Unit time cost per step; constant memory cost
  - You control how much memory is used

- RandomRestart(TabuSearch (LocalSearch()))

Tabu search wrapper (__inside__ random restart!)

New State → FIFO QUEUE → Oldest State

New State → HASH TABLE → State Present?
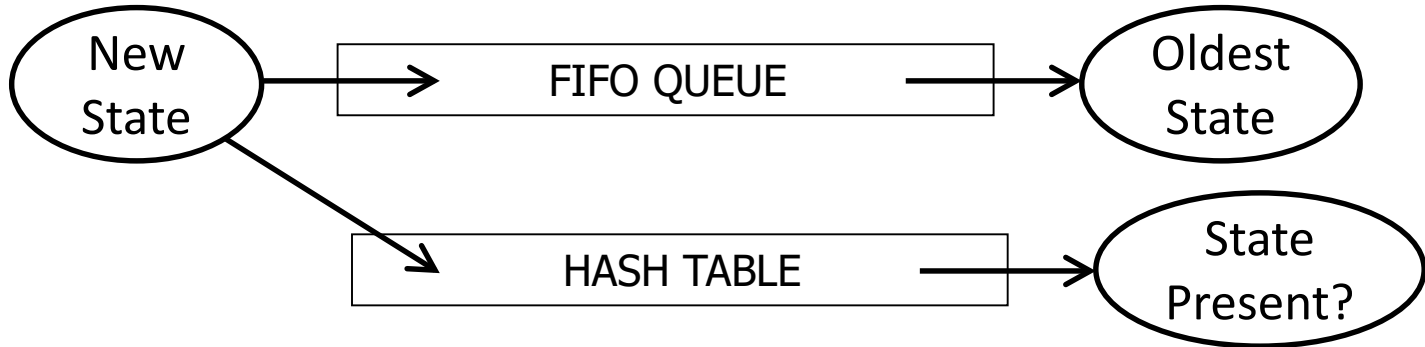
$best\_found \leftarrow current\_state \leftarrow$ **RandomState**() // initialize
loop do // now do local search
  if (tired of doing it) **then return** $best\_found$ **else**
    $neighbor \leftarrow$ **MakeNeighbor**($current\_state$)
    if ($neighbor$ is in $hash\_table$) **then** discard $neighbor$
    else push $neighbor$ onto $fifo$, pop $oldest\_state$
      remove $oldest\_state$ from $hash\_table$, insert $neighbor$
      $current\_state \leftarrow neighbor$;
      if (**Cost**($current\_state$) < **Cost**($best\_found$))
   **then** $best\_found \leftarrow current\_state$

Local search algorithms

- Hill-climbing search
  - Gradient descent in continuous state spaces
  - Can use, e.g., Newton's method to find roots
- Simulated annealing search
- Local beam search
- Genetic algorithms
- Linear Programming (for specialized problems)

Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the search space increases to high dimensionality.

- <u>Problems:</u> depending on state, can get stuck in local maxima
  - Many other problems also endanger your success!!
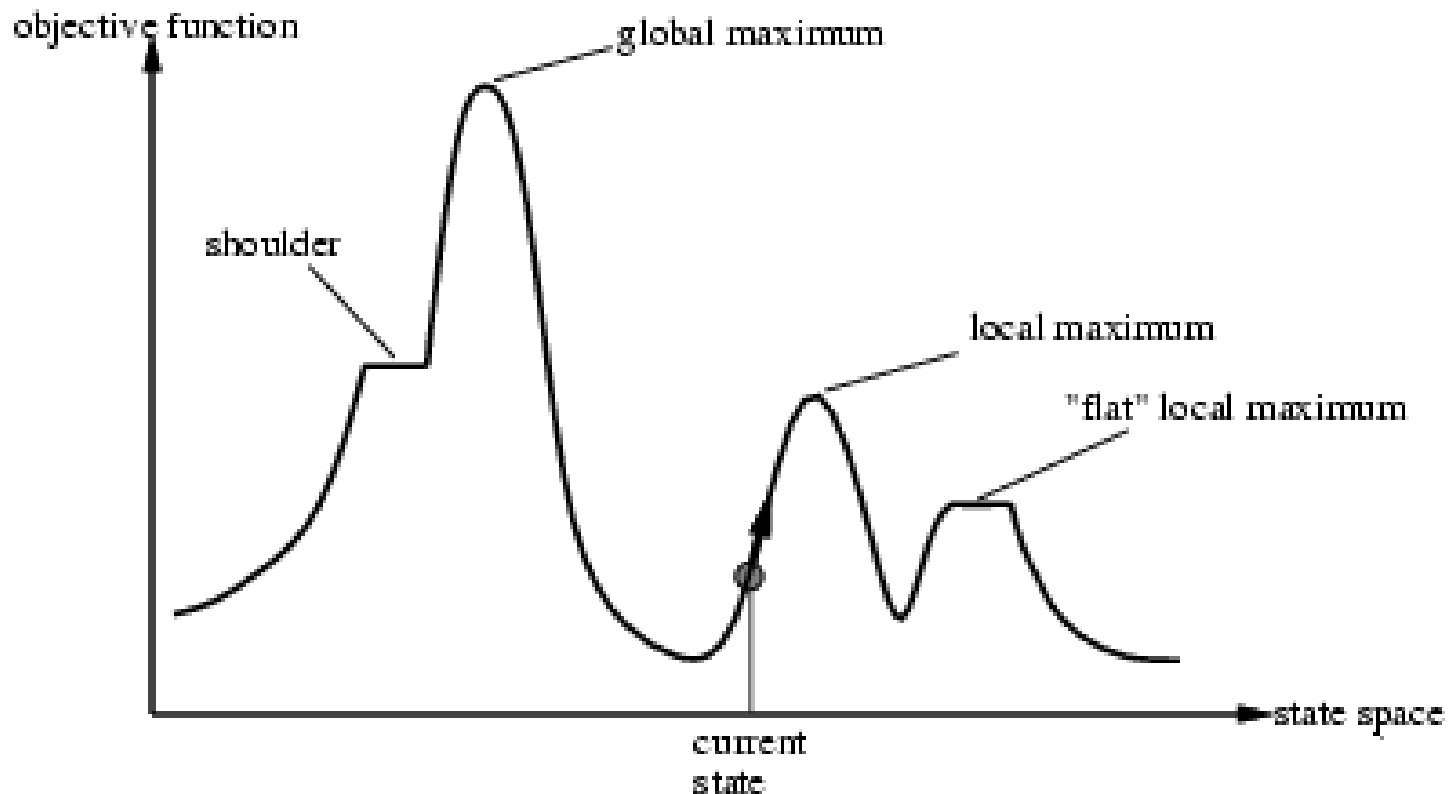
Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the search space increases to high dimensionality.

- <u>Ridge problem:</u> Every neighbor appears to be downhill
  - But the search space has an uphill!! (worse in high dimensions)

<u>Ridge:</u>
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.

Figure 4.4 FILES: figures/ridge.eps (Tue Nov 3 16:23:29 2009). Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Hill-climbing search

You must shift effortlessly between maximizing value and minimizing cost

"...like trying to find the top of Mount Everest in a thick fog while suffering from amnesia"

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
    inputs: *problem*, a problem
    local variables: *current*, a node
                      neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[*problem*])
loop do
    neighbor ← a highest-valued successor of *current*
    if VALUE[neighbor] $\leq$ VALUE[current] **then return** STATE[*current*]
    current ← *neighbor*

Equivalently:
"...a lowest-cost successor..."

Equivalently: "if **COST**[neighbor] $\geq$ **COST**[current] then ..."

Simulated annealing (Physics!)

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

-

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state
   inputs: *problem*, a problem
          schedule, a mapping from time to "temperature"
   local variables: *current*, a node
                 next, a node
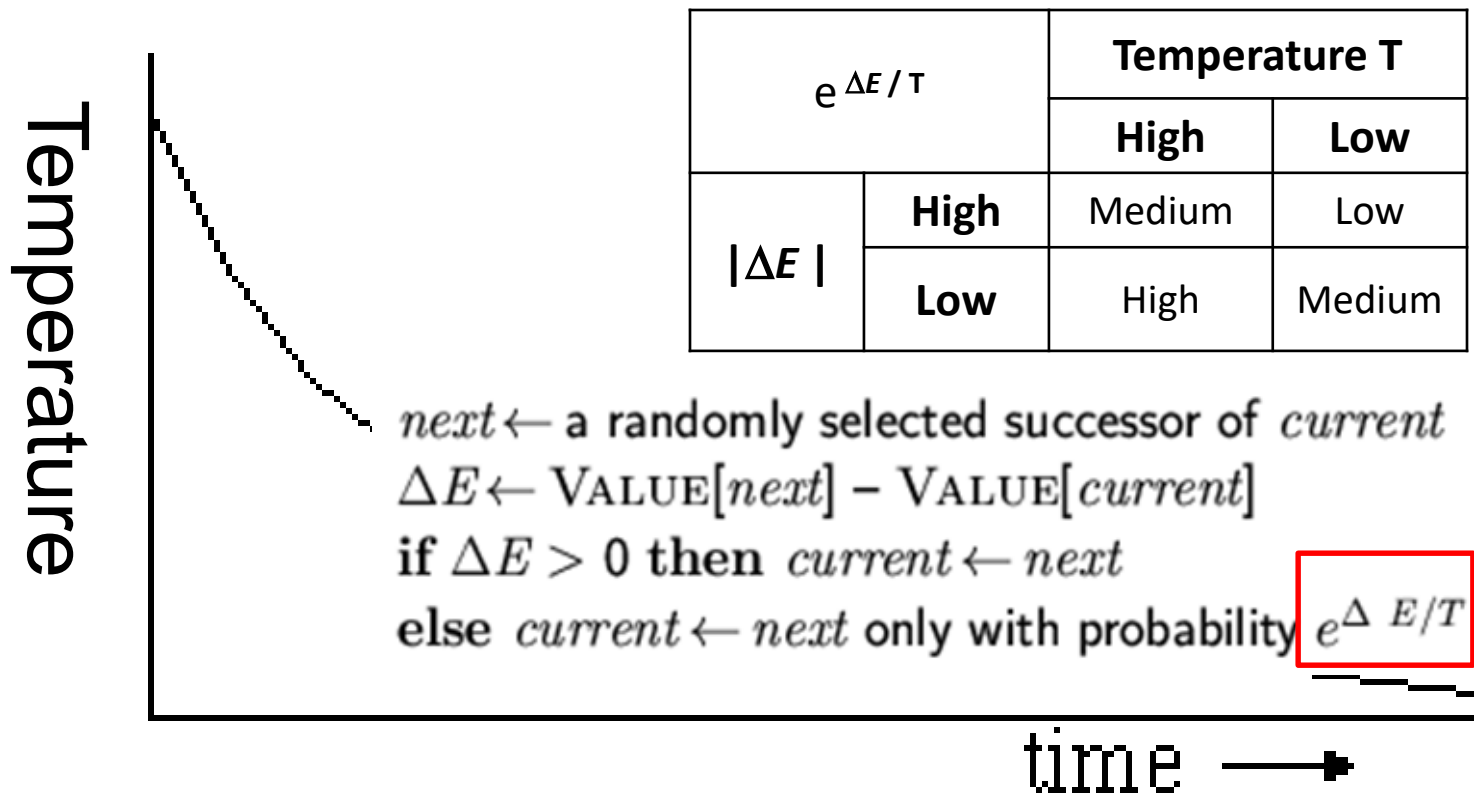 T, a "temperature" controlling prob. of downward steps

current ← MAKE-NODE(INITIAL-STATE[*problem*])
for $t \leftarrow$ **1 to** ∞ **do**
   $T \leftarrow schedule[t]$
   if $T = 0$ **then return** *current*
   next ← a randomly selected successor of *current*
   $\Delta E \leftarrow$ VALUE[*next*] − VALUE[*current*]
   if $\Delta E > 0$ **then** *current* ← *next*
   else *current* ← *next* only with probability $e^{\Delta E/T}$

Improvement: Track the BestResultFoundSoFar. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.
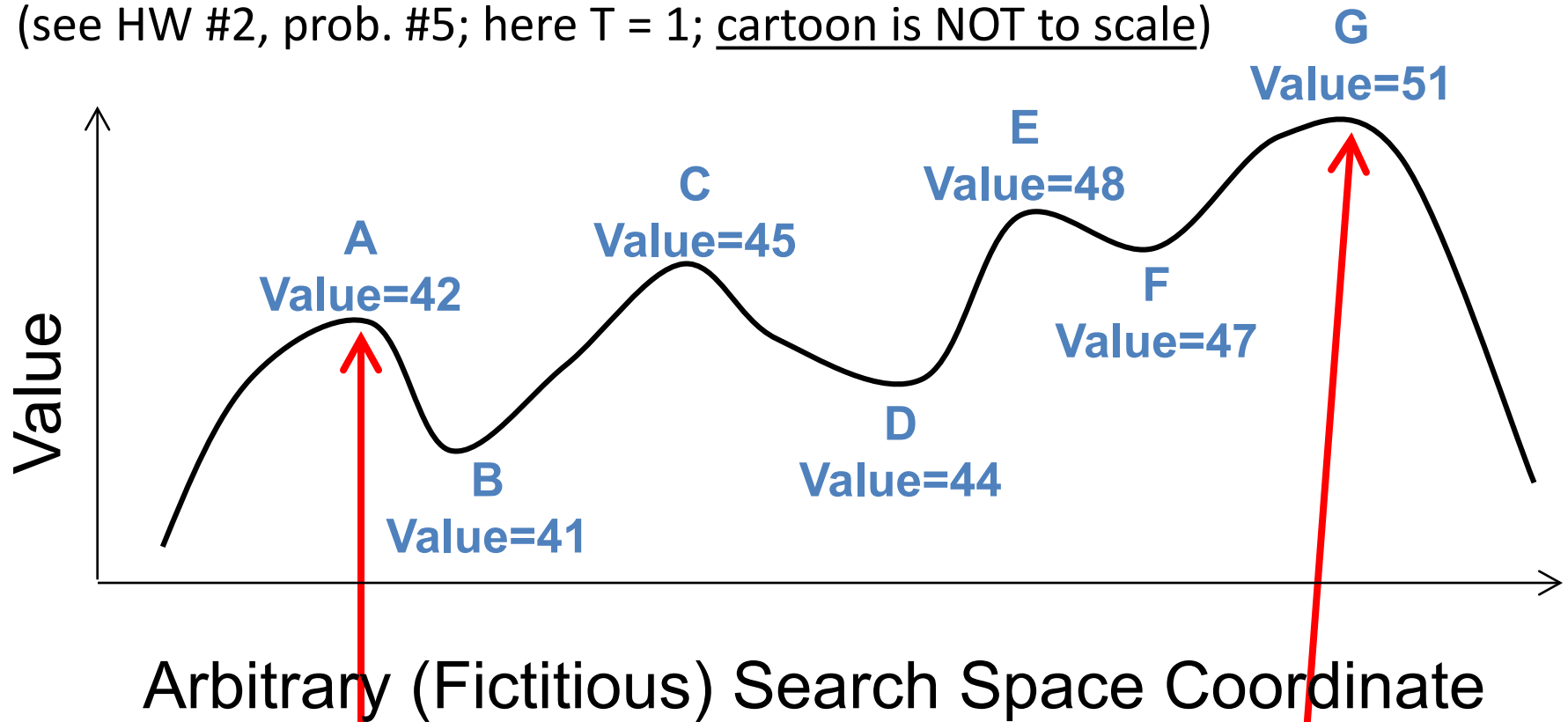
Probability(accept worse successor)

- Decreases as temperature T decreases
- Increases as |Δ E| decreases
- Sometimes, step size also decreases with T

(accept very bad moves early on; later, mainly accept "not very much worse")

$e^{\Delta E / T}$		Temperature T	
		High	**Low**
|ΔE|	**High**	Medium	Low
	Low	High	Medium

$next \leftarrow$ a randomly selected successor of $current$

$\Delta E \leftarrow$ VALUE$[next]$ – VALUE$[current]$

if $\Delta E > 0$ then $current \leftarrow next$

else $current \leftarrow next$ only with probability $e^{\Delta E/T}$

Temperature (y-axis) vs. **time →** (x-axis)

Goal: "ratchet up" a jagged slope

A
Value=42
$\Delta E(AB)$=-1
$P(AB) \approx .37$

C
Value=45
$\Delta E(CB)$=-4
$\Delta E(CD)$=-1
$P(CB) \approx .018$
$P(CD) \approx .37$

E
Value=48
$\Delta E(ED)$=-4
$\Delta E(EF)$=-1
$P(ED) \approx .018$
$P(EF) \approx .37$

G
Value=51
$\Delta E(GF)$=-4
$P(GF) \approx .018$

B
Value=41
$\Delta E(BA)$=1
$\Delta E(BC)$=4
$P(BA)$=1
$P(BC)$=1

D
Value=44
$\Delta E(DC)$=1
$\Delta E(DE)$=4
$P(DC)$=1
$P(DE)$=1

F
Value=47
$\Delta E(FE)$=1
$\Delta E(FG)$=4
$P(FE)$=1
$P(FG)$=1

Your "random restart wrapper" starts here.
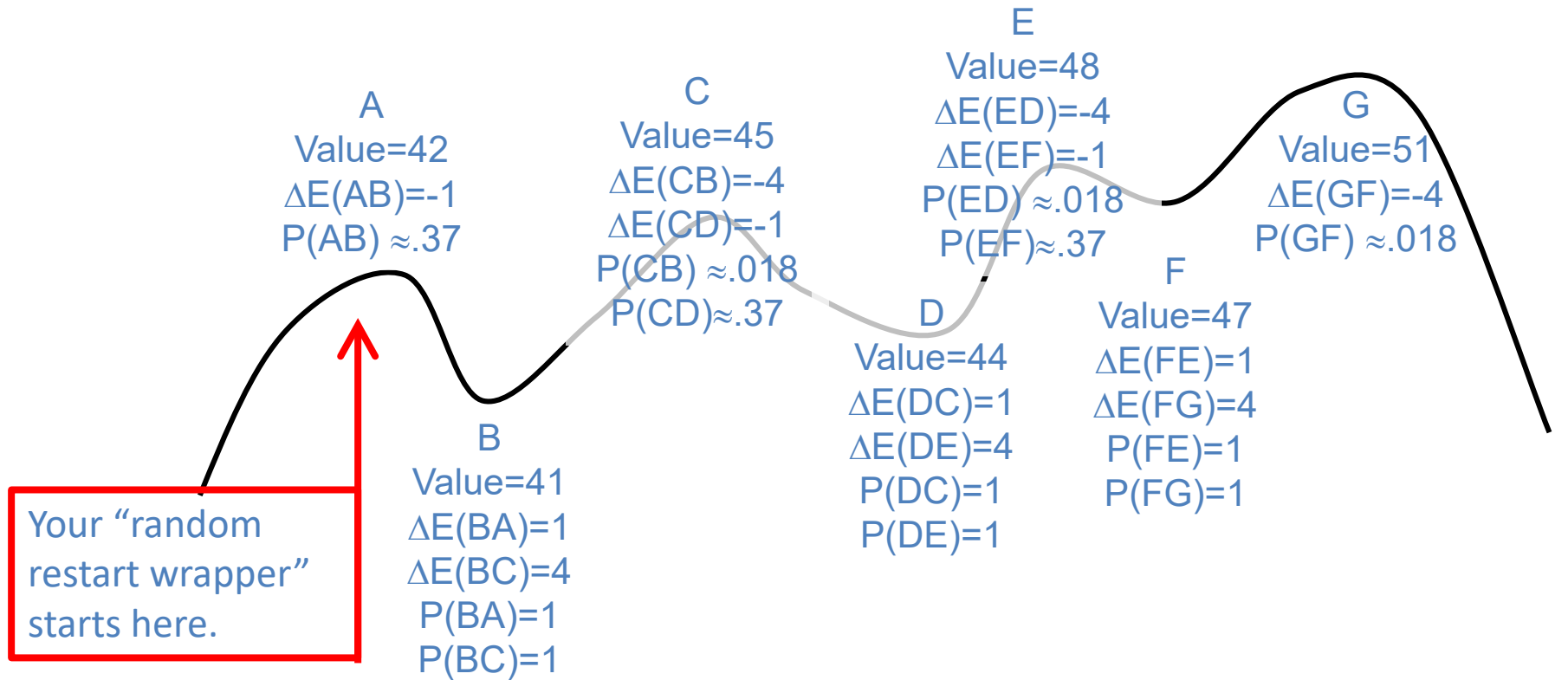
x	-1	-4
e^x	$\approx .37$	$\approx .018$

From A you will accept a move to B with $P(AB) \approx .37$.
From B you are equally likely to go to A or to C.
From C you are $\approx$20X more likely to go to D than to B.
From D you are equally likely to go to C or to E.
From E you are $\approx$20X more likely to go to F than to D.
From F you are equally likely to go to E or to G.
Remember best point you ever found (G or neighbor?).

This is an illustrative *cartoon*…

Local beam search

- Keep track of *k* states rather than just one

- Start with *k* randomly generated states

- At each iteration, all the successors of all *k* states are generated

- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

- Concentrates search effort in areas believed to be fruitful
 - May lose diversity as search progresses, resulting in wasted effort

Local beam search

Create *k* random initial states

Generate their children

Select the *k* best children

Repeat indefinitely…
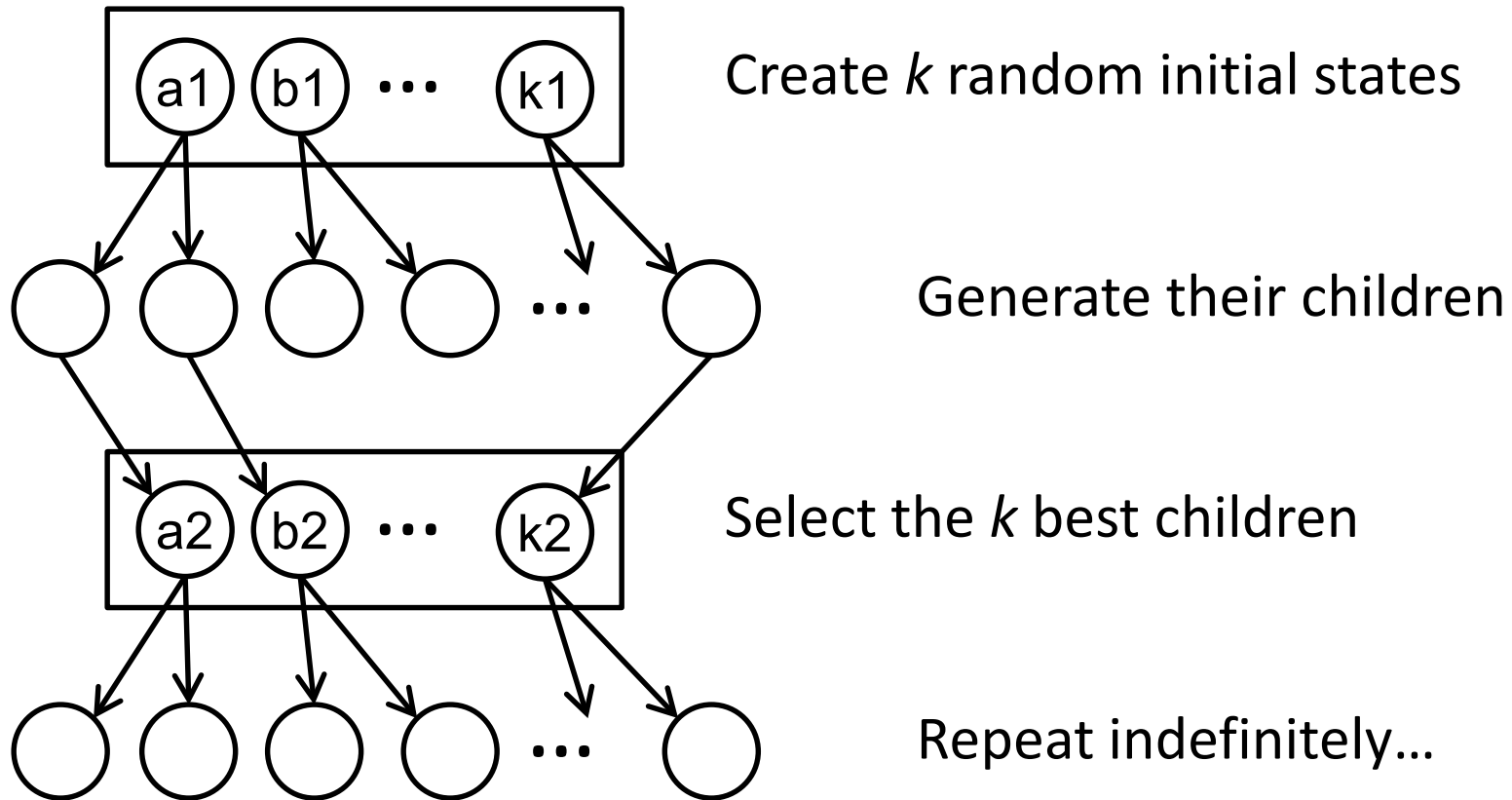
Is it better than simply running *k* searches?
Maybe…??

Genetic algorithms (**Darwin!!**)

- A state = a string over a finite alphabet (an **individual**)
  - A successor state is generated by combining two parent states

- Start with k randomly generated states (a **population**)

- **Fitness** function (= our heuristic objective function).
  - Higher fitness values for better states.

- **Select** individuals for next generation based on fitness
  - P(individual in next gen.) = individual fitness/total population fitness

- **Crossover** fit parents to yield next generation (**offspring**)

- **Mutate** the offspring randomly with some low probability

Genetic algorithms

(a) Initial Population | (b) Fitness Function | (c) Selection | (d) Cross-Over | (e) Mutation
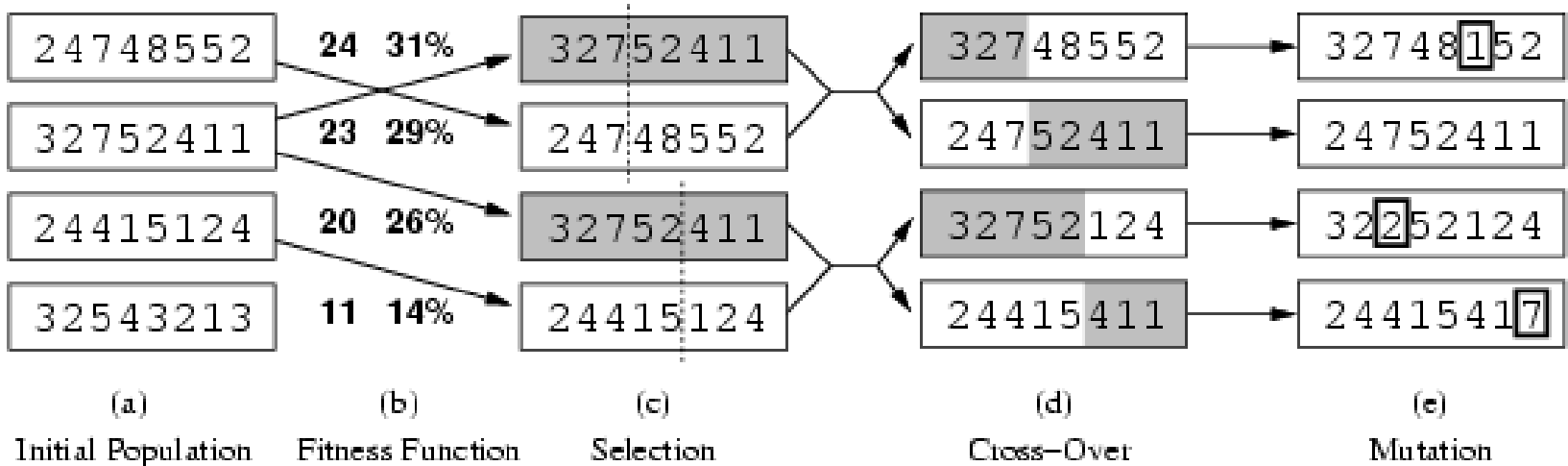
- Fitness function (value): number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
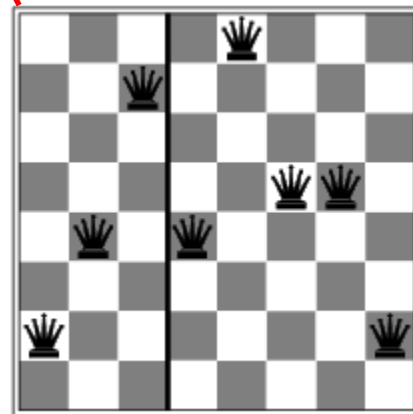- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29%; etc.

(a)	(b)	(c)	(d)	(e)
Initial Population	Fitness Function	Selection	Cross-Over	Mutation
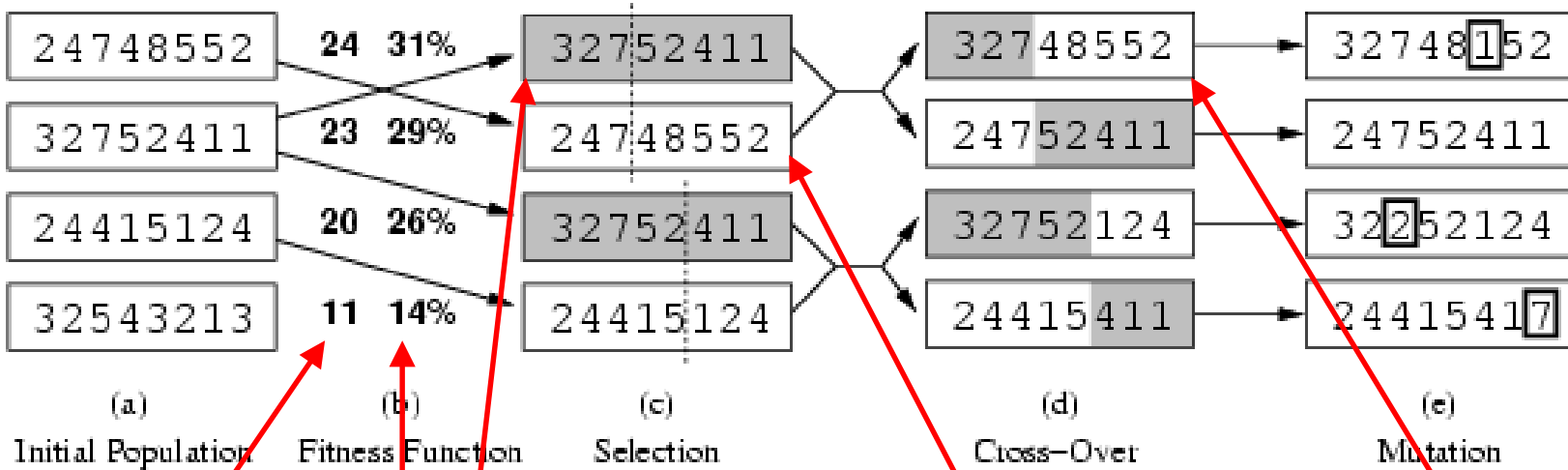
fitness = #non-attacking queens

probability of being in next generation = fitness/(Σ_i fitness_i)

How to convert a fitness value into a probability of being in the next generation.
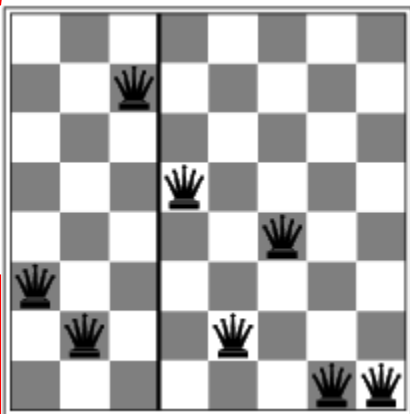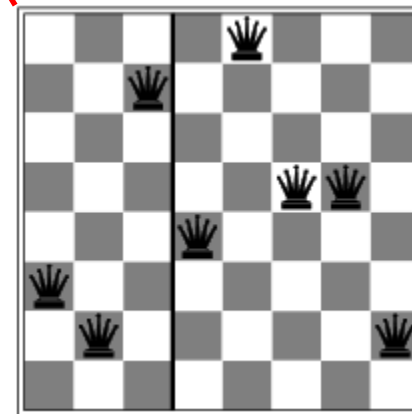
- Fitness function: #non-attacking queen pairs
  - min = 0, max = 8 × 7/2 = 28
- Σ_i fitness_i = 24+23+20+11 = 78
- P(child_1 in next gen.) = fitness_1/(Σ_i fitness_i) = 24/78 = 31%
- P(child_2 in next gen.) = fitness_2/(Σ_i fitness_i) = 23/78 = 29%; etc

Review Adversarial (Game) Search Chapter 5.1-5.4

- Minimax Search with Perfect Decisions (5.2)
  - Impractical in most cases, but theoretical basis for analysis
- Minimax Search with Cut-off (5.4)
  - Replace terminal leaf utility by heuristic evaluation function
- Alpha-Beta Pruning (5.3)
  - The fact of the adversary leads to an advantage in search!
- Practical Considerations (5.4)
 - Redundant path elimination, look-up tables, etc.

Games as Search

- Two players: MAX and MIN

- MAX moves first and they take turns until the game is over
  - Winner gets reward, loser gets penalty.
  - "Zero sum" means the sum of the reward and the penalty is a constant.

- Formal definition as a search problem:
  - **Initial state:** Set-up specified by the rules, e.g., initial board configuration of chess.
  - **Player(s):** Defines which player has the move in a state.
  - **Actions(s):** Returns the set of legal moves in a state.
  - **Result(s,a):** Transition model defines the result of a move.
  - (**2nd ed.: Successor function:** list of (move,state) pairs specifying legal moves.)
  - **Terminal-Test(s):** Is the game finished? True if finished, false otherwise.
  - **Utility function(s,p):** Gives numerical value of terminal state s for player p.
    - E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    - E.g., win (+1), lose (0), and draw (1/2) in chess.

- MAX uses search tree to determine "best" next move.

An optimal procedure:
The Min-Max method

Will find the <u>optimal strategy and best next move </u>for Max:

- 1. Generate the whole game tree, down to the leaves.

- 2. Apply utility (payoff) function to each leaf.

- 3. Back-up values from leaves through branch nodes:
  – a Max node computes the Max of its child values
  – a Min node computes the Min of its child values

- 4. At root: choose move leading to the child of highest value.

Two-ply Game Tree

MAX

The minimax decision → 3

MIN

3 · 3   2   2

3 · 12 · 8   2 · 4 · 6   14 · 5 · 2
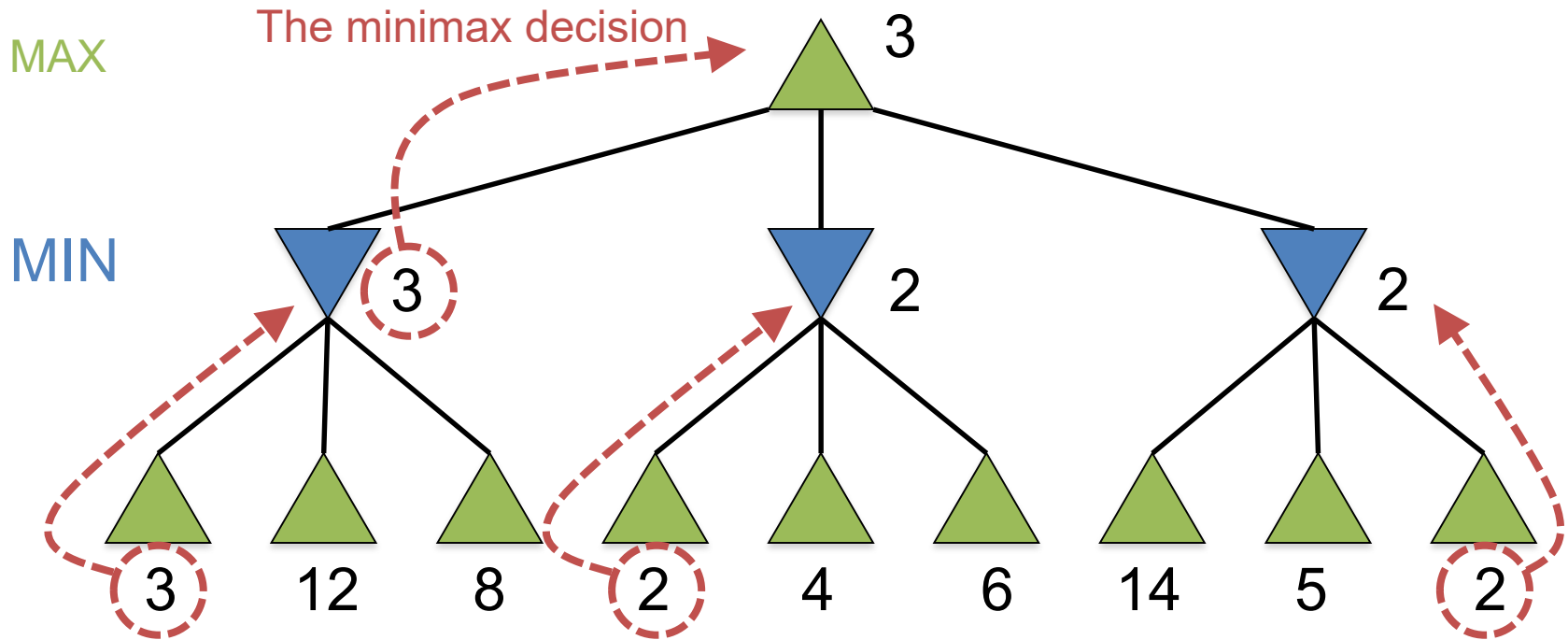
Minimax maximizes the utility of the worst-case outcome for MAX

Pseudocode for Minimax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  inputs: *state*, current state in game
  return arg max$_{a \in \text{ACTIONS}(state)}$ MIN-VALUE(Result(*state,a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a* in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v,MIN-VALUE(Result(*state,a*)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for *a* in ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v,MAX-VALUE(Result(*state,a*)))
  return v

Properties of minimax

- <u>Complete?</u>
  - Yes (if tree is finite).

- <u>Optimal?</u>
  - Yes (against an optimal opponent).
  - Can it be beaten by an opponent playing sub-optimally?
    - No. (Why not?)

- <u>Time complexity?</u>
  - $O(b^m)$

- <u>Space complexity?</u>
  - $O(bm)$ (depth-first search, generate all actions at once)
  - $O(m)$ (backtracking search, generate actions one at a time)

Cutting off search

MINIMAXCUTOFF is identical to MINIMAXVALUE except
1. TERMINAL? is replaced by CUTOFF?
2. UTILITY is replaced by EVAL

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

Static (Heuristic) Evaluation Functions

- **An Evaluation Function:**
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

- If the board evaluation is X for a player, it's -X for the opponent
  - "Zero-sum game"

Evaluation functions
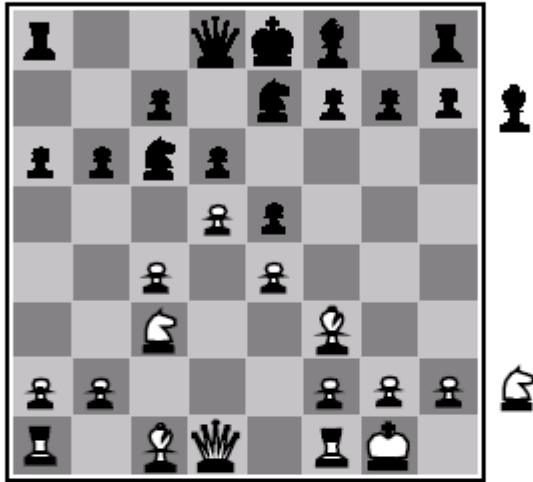
Black to move

White slightly better
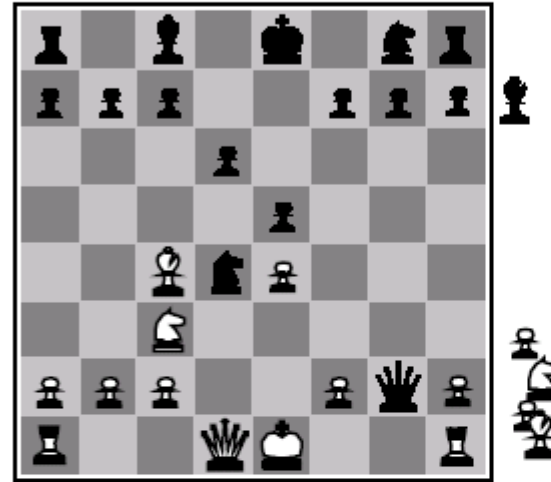
White to move

**Black winning**

For chess, typically *linear* weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) $-$ (number of black queens), etc.

General alpha-beta pruning

- Consider a node *n* in the tree ---

- If player has a better choice at:
    - Parent node of n
    - Or any choice point further up

- Then *n* will never be reached in play.

- Hence, when that much is known about *n*, it can be pruned.
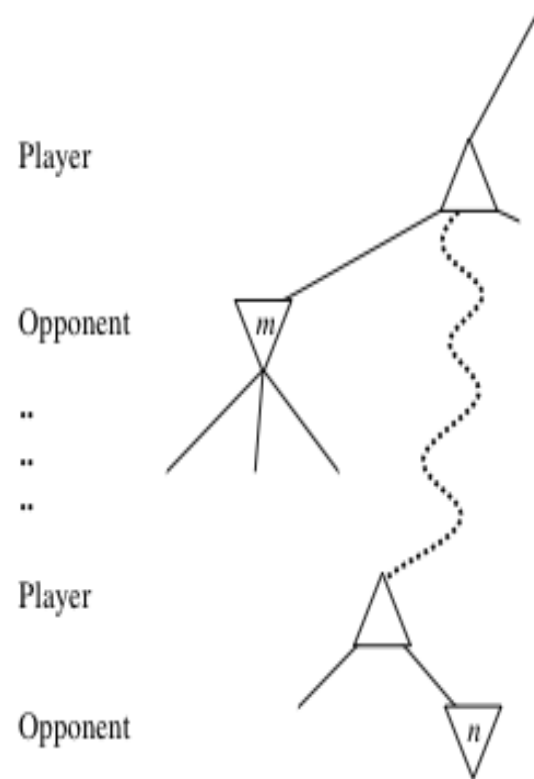
Alpha-beta Algorithm

- Depth first search
  - only considers nodes along a single path from root at any time

α = highest-value choice found at any choice point of path for MAX
       (initially, α = –infinity)

β = lowest-value choice found at any choice point of path for MIN
       (initially, β = +infinity)

- Pass current values of α and β down to child nodes during search.
- Update values of α and β during search:
  - MAX updates α at MAX nodes
  - MIN updates β at MIN nodes
- Prune remaining branches at a node when $\alpha \geq \beta$

Recursive α-β pruning: R&N Fig. 5.7

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty$, $+\infty$)
  return the *action* in ACTIONS(*state*) with value v

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(*state*,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(*state*,a), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Simple stub to call recursion functions
Initialize alpha, beta; get best value
Score each action; return best action

If Cutoff reached, return Eval heuristic
Otherwise, find our best child:
If our options become too good, our min
  ancestor will never let us come this way,
  so prune now & return best value so far
Finally, return the best value we found

If Cutoff reached, return Eval heuristic
Otherwise, find our worst child:
If our options become too bad, our max
  ancestor will never let us come this way,
  so prune now & return worst value so far
Finally, return the worst value we found

Recursive α-β pruning variant: Prune when α ≥ β

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    if CUTOFF-TEST(state) then return EVAL(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(state,a) , α, β))
        α ← MAX(α, v)
        if α ≥ β then return v
    return v

function MIN-VALUE(state, α, β) returns a utility value
    if CUTOFF-TEST(state) then return EVAL(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(state,a) , α, β))
        β ← MIN(β, v)
        if α ≥ β then return v
    return v
```

This variant has a conceptually simpler pruning rule (α ≥ β), but when pruning occurs it makes one extra call to MAX(). Both variants yield the same pruning behavior, and **both are considered correct on tests.**

When to Prune?

- **<u>Prune whenever $\alpha \geq \beta$.</u>**

  – Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.

    - **<u>Max nodes update alpha</u>** based on children's returned values.

  – Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.

    - **<u>Min nodes update beta</u>** based on children's returned values.

α/β Pruning vs. Returned Node Value

- Some students are confused about the use of α/β pruning vs. the returned value of a node

- α/β are used **ONLY FOR PRUNING**
  - α/β have no effect on anything other than pruning
  - IF (α >= β) THEN prune & return current node value

- Returned node value = "best" child seen so far
  - Maximum child value seen so far for MAX nodes
  - Minimum child value seen so far for MIN nodes
  - If you prune, return to parent "best" child so far

- Returned node value is received by parent

Alpha-Beta Example

Max calculates the same node value, and makes the same move!
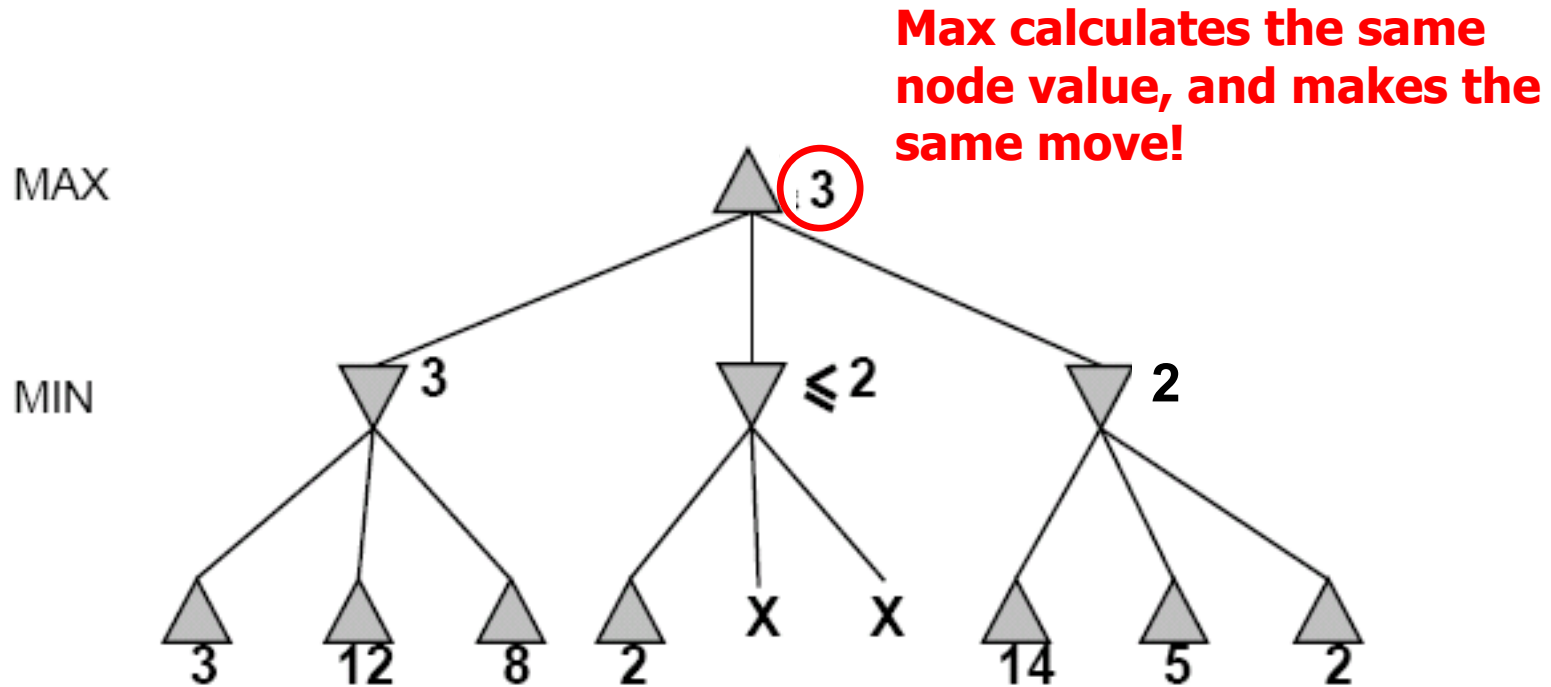
Review Detailed Example of Alpha-Beta Pruning in lecture slides.

Review Constraint Satisfaction
R&N 6.1-6.4 (except 6.3.3)

- What is a CSP?

- Backtracking search for CSPs
    - Choose a variable, then choose an order for values
    - Minimum Remaining Values (MRV), Degree Heuristic (DH), Least Constraining Value (LCV)

- Constraint propagation
    - Forward Checking (FC), Arc Consistency (AC-3)

- Local search for CSPs
    - Min-conflicts heuristic

Constraint Satisfaction Problems

- What is a CSP?
  - Finite set of variables, X_1, X_2, …, X_n
  - Nonempty domain of possible values for each: D_1, …, D_n
  - Finite set of constraints, C_1, …, C_m
    - Each constraint C_i limits the values that variables can take, e.g., $X_1 \neq X_2$
  - Each constraint C_i is a pair: C_i = (*scope*, *relation*)
    - Scope = tuple of variables that participate in the constraint
    - Relation = list of allowed combinations of variables
      May be an explicit list of allowed combinations
      May be an abstract relation allowing membership testing & listing

- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain-specific expertise required)

CSPs --- what is a solution?

- A *__state__* is an *__assignment__* of values to some variables.
  - *__Complete__* assignment
    - = every variable has a value.
  - *__Partial__* assignment
    - = some variables have no values.
  - *__Consistent__* assignment
    - = assignment does not violate any constraints

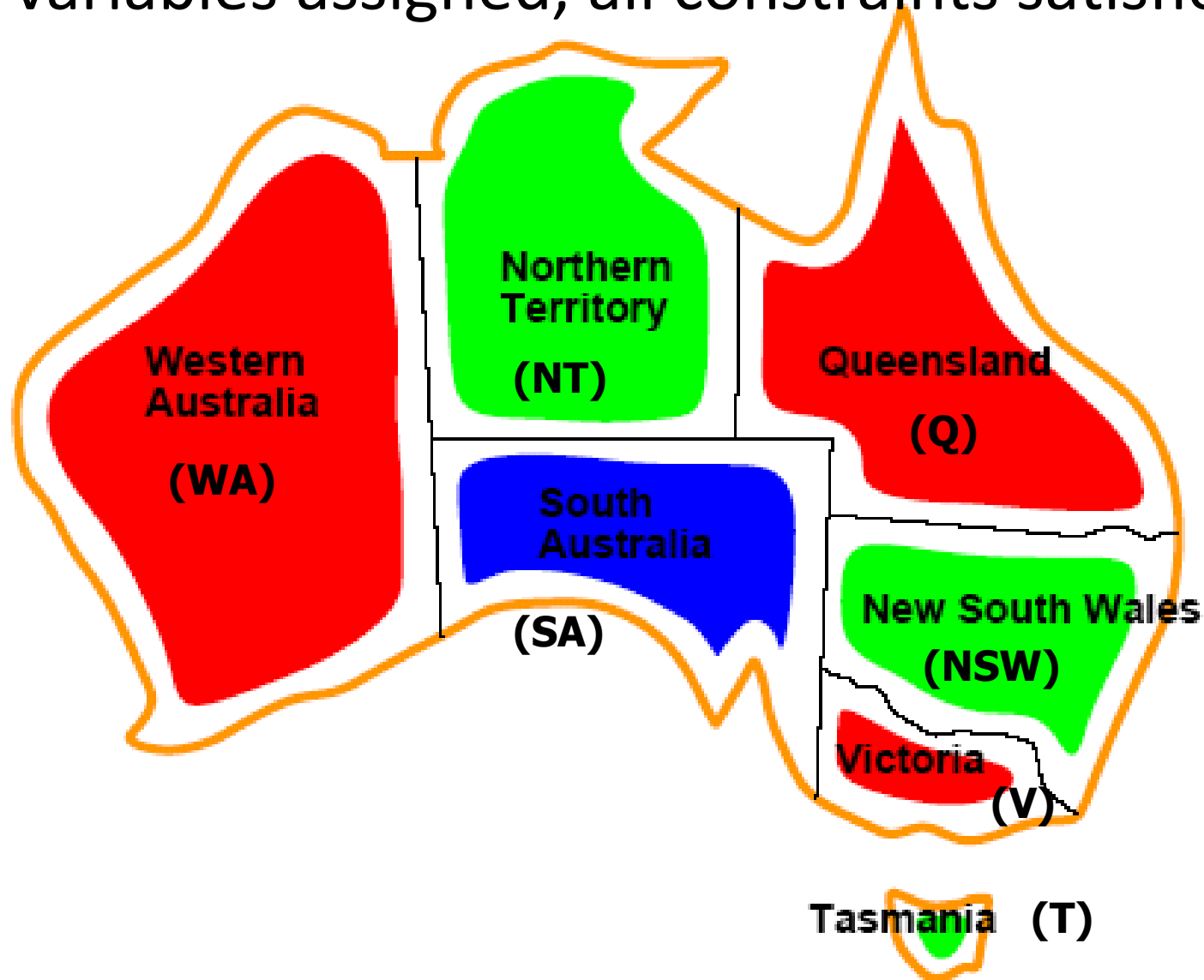- A *__solution__* is a *__complete__* and *__consistent__* assignment.

CSP example: map coloring
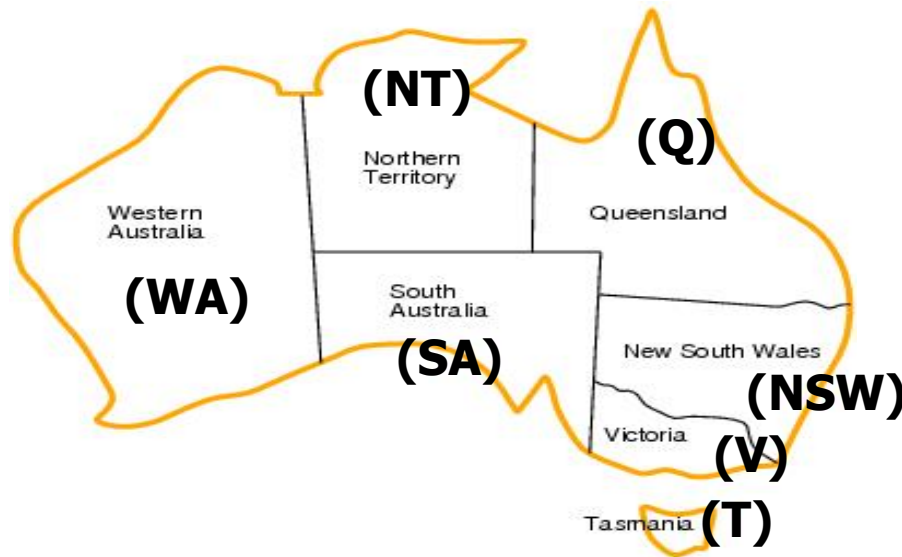
- **Variables:** *WA, NT, Q, NSW, V, SA, T*
- **Domains:** $D_i = \{red, green, blue\}$
- **Constraints:** Adjacent regions must have different colors, e.g., *WA $\neq$ NT*.

Example: Map coloring solution

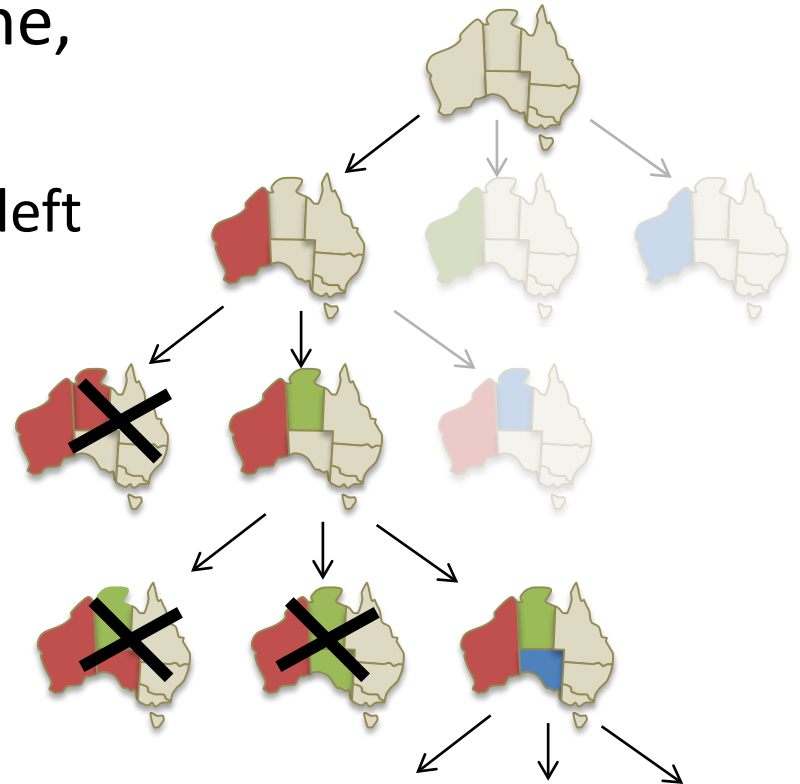All variables assigned, all constraints satisfied.

Example: Map Coloring

- **Constraint graph**
  - Vertices: variables
  - Edges: constraints
    (connect involved variables)

- **Graphical model**
  - Abstracts the problem to a canonical form
  - Can reason about problem through graph connectivity
  - Ex: Tasmania can be solved independently (more later)

- **Binary CSP**
  - Constraints involve at most two variables
  - Sometimes called "pairwise"
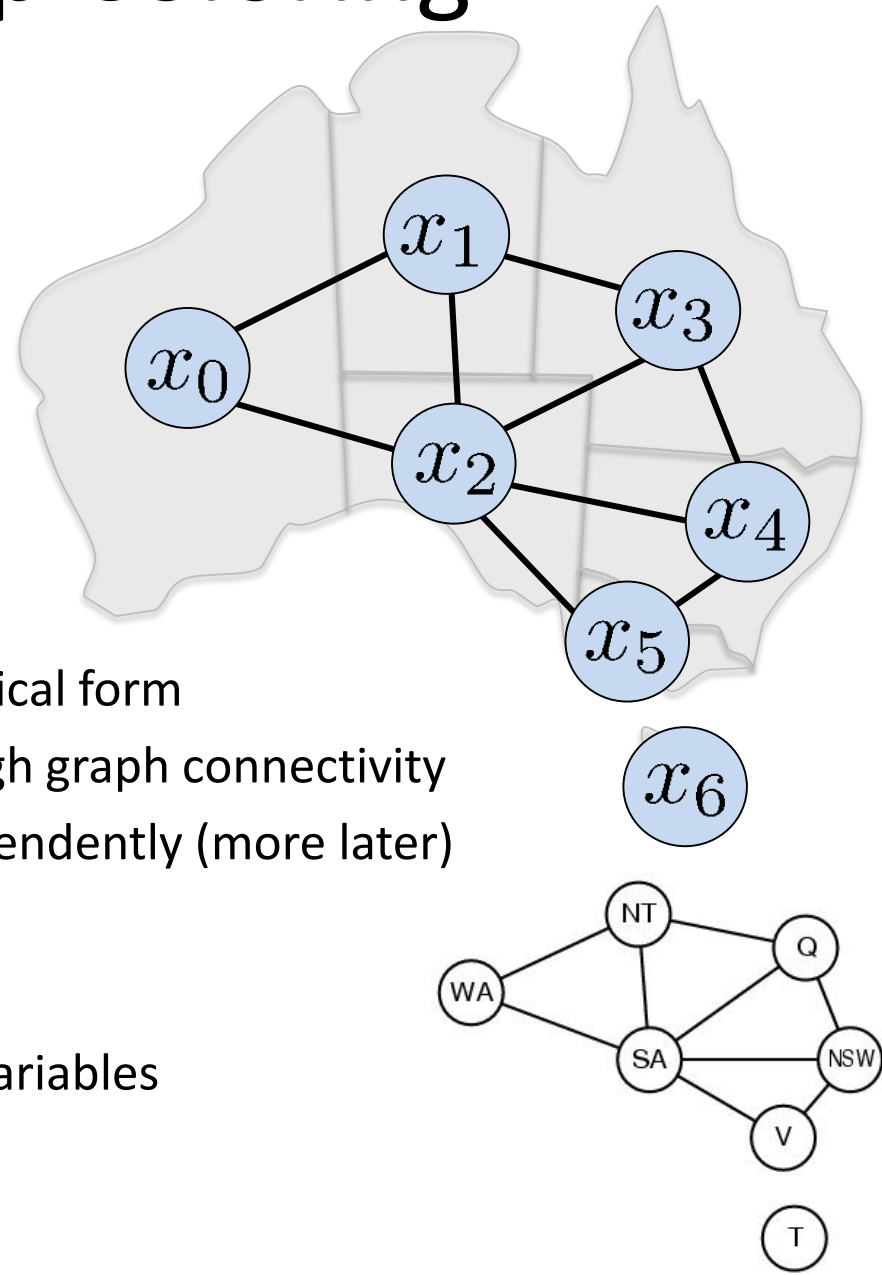
Backtracking search

- Similar to depth-first search
  - At each level, pick a single variable to expand
  - Iterate over the domain values of that variable

- Generate children one at a time,
  - One child per value
  - Backtrack when no legal values left

- Uninformed algorithm
  - Poor general performance

Backtracking search (Figure 6.5)

function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure
    return RECURSIVE-BACKTRACKING({} , *csp*)

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** a solution or failure
    if *assignment* is complete **then return** *assignment*
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment,csp*)
    for each *value* **in** ORDER-DOMAIN-VALUES(*var, assignment, csp*) **do**
        if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
            add *{var=value}* to assignment
            result ← RRECURSIVE-BACTRACKING(*assignment, csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var=value}* from *assignment*
    return *failure*

Minimum remaining values (MRV)

var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment,csp*)

- A.k.a. most constrained variable heuristic

- *Heuristic Rule*: choose variable with the fewest legal moves
  - e.g., will immediately detect failure if X has no legal values

Degree heuristic for the initial variable

- *Heuristic Rule*: select variable that is involved in the largest number of constraints on other unassigned variables.

- Degree heuristic can be useful as a tie breaker.

- *In what order should a variable's values be tried?*

Backtracking search (Figure 6.5)

function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure

    return RECURSIVE-BACKTRACKING(*{}* , *csp*)

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** a solution or failure

    if *assignment* is complete **then return** *assignment*

    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

    for each *value* **in** ORDER-DOMAIN-VALUES(*var, assignment, csp*) **do**

        if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**

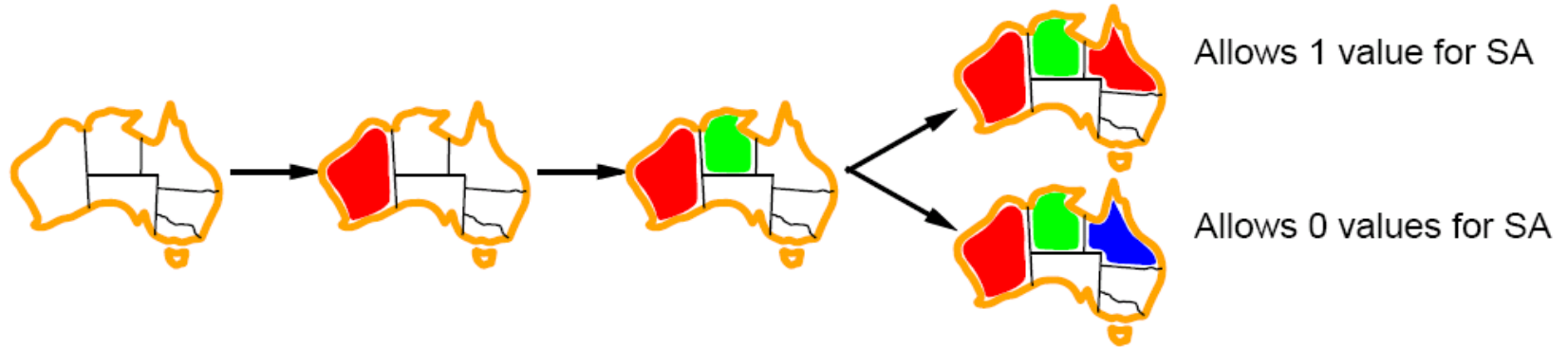   add *{var=value}* to assignment

            result ← RRECURSIVE-BACTRACKING(*assignment, csp*)

            if *result* $\neq$ *failure* **then return** *result*

            remove *{var=value}* from *assignment*

    return *failure*

Least constraining value for value-ordering

Allows 1 value for SA

Allows 0 values for SA

- Least constraining value heuristic

- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments

Look-ahead: Constraint propagation

- **Intuition:**
  - Some domains have values that are <u>inconsistent</u> with the values in some other domains
  - Propagate constraints to remove inconsistent values
  - Thereby reduce future branching factors
- **Forward checking**
  - Check each unassigned neighbor in constraint graph
- **Arc consistency (AC-3 in R&N)**
  - Full arc-consistency everywhere until quiescence
  - Can run as a preprocessor
    - Remove obvious inconsistencies
  - Can run after each step of backtracking search
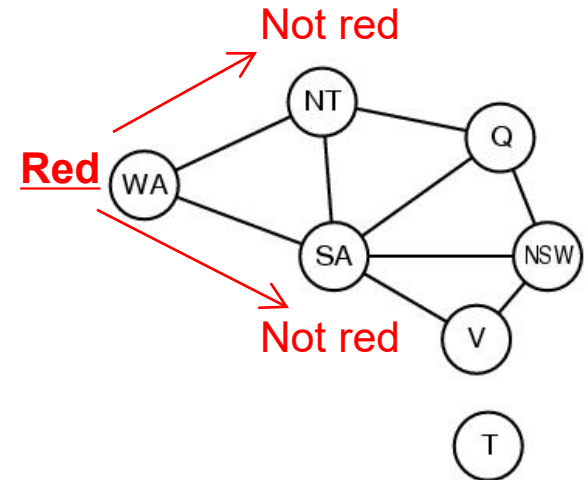   - Maintaining Arc Consistency (MAC)

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - ONLY check neighbors of most recently assigned variable

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - ONLY check neighbors of most recently assigned variable

Not red

NT

Q

Red WA

SA

NSW

Not red

V

T

WA	NT	Q	NSW	V	SA	T
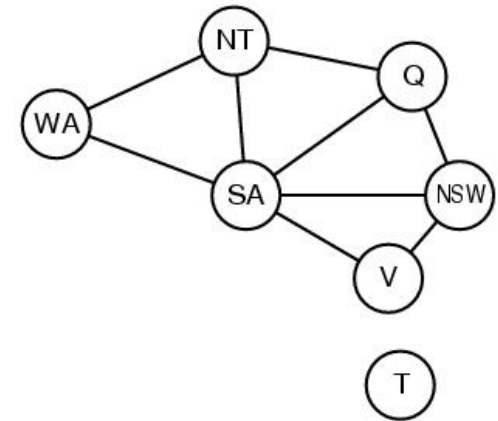
Assign {WA = red}
Effect on other variables (neighbors of WA):
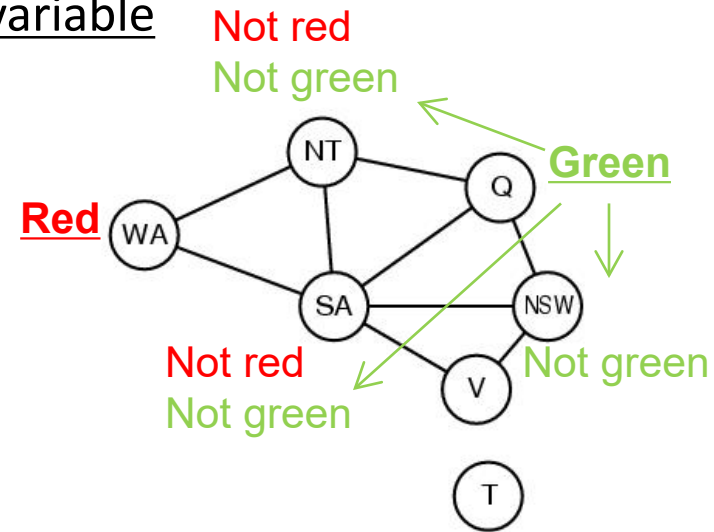- NT can no longer be red
- SA can no longer be red

Forward checking

- Idea:
    - Keep track of remaining legal values for unassigned variables
    - Backtrack when any variable has no legal values
    - Check neighbors of <u>most recently assigned variable</u>

Assign {Q = green}
Effect on other variables (neighbors of Q):
- NT can no longer be green
- SA can no longer be green
- NSW can no longer be green

(We already have failure, but FC is too simple to detect it now)

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - Check neighbors of <u>most recently assigned variable</u>

Assign {V = blue}
Effect on other variables (neighbors of V):
- NSW can no longer be blue
- SA can no longer be blue **(no values possible!)**

Forward checking has detected that this partial assignment is inconsistent with any complete assignment

Arc consistency (AC-3) algorithm

- An Arc $X \rightarrow Y$ is consistent iff for *every* value x of X there is *some* value y of Y that is consistent with x

- Put all arcs $X \rightarrow Y$ on a queue
  - Each undirected constraint graph arc is two directed arcs
  - Undirected X—Y becomes directed $X \rightarrow Y$ and $Y \rightarrow X$
  - $X \rightarrow Y$ and $Y \rightarrow X$ both go on queue, separately

- Pop one arc $X \rightarrow Y$ and remove any inconsistent values from X

- If any change in X, put all arcs $Z \rightarrow X$ back on queue, where Z is any neighbor of X that is not equal to Y

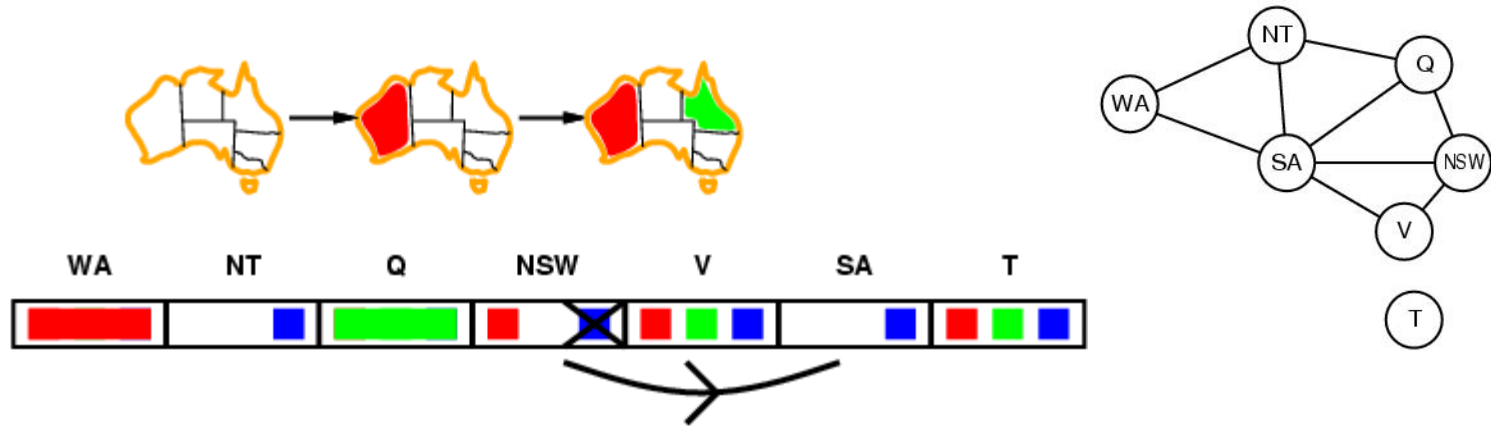- Continue until queue is empty

Arc consistency (AC-3)

- Simplest form of propagation makes each arc consistent

- $X \rightarrow Y$ is consistent iff (iff = if and only if)

  for every value x of X there is some allowed value y for Y *(note: directed!)*
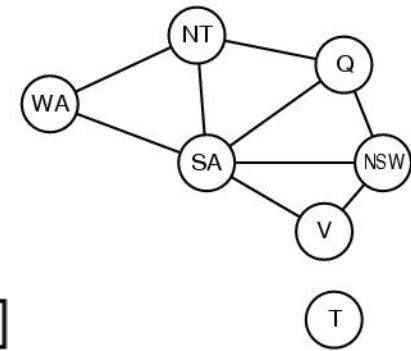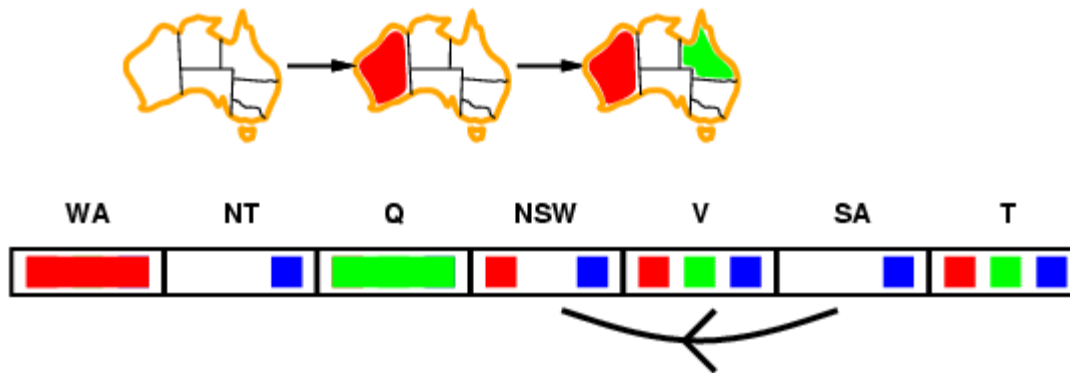
- Consider state after WA=red, Q=green

  - SA $\rightarrow$ NSW is consistent because

    SA = blue and NSW = red satisfies all constraints on SA and NSW

Arc consistency

- Simplest form of propagation makes each arc consistent

- $X \rightarrow Y$ is consistent iff

    for every value *x* of *X* there is some allowed value *y for Y*   *(note: directed!)*

- Consider state after WA=red, Q=green
    - NSW $\rightarrow$ SA consistent if

        NSW = red and SA = blue
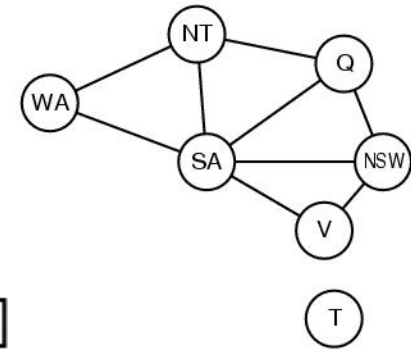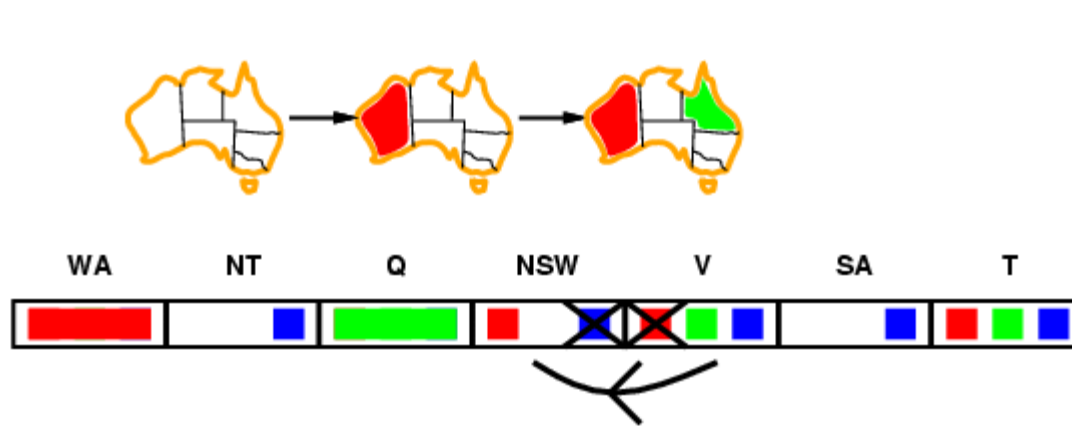
        NSW = blue and SA = ???   => NSW = blue can be pruned
                                      No current domain value for SA is consistent

If *X* loses a value, neighbors of *X* need to be rechecked

Arc consistency

- Simplest form of propagation makes each arc consistent

- $X \rightarrow Y$ is consistent iff

    for every value *x* of *X* there is some allowed value *y for Y*   *(note: directed!)*
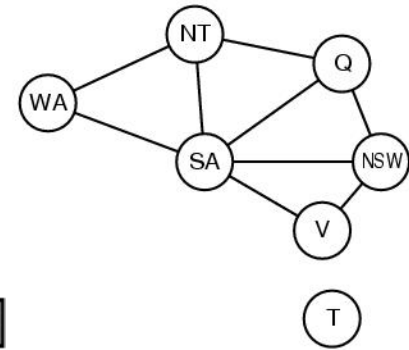

- **Enforce arc consistency:**

    – arc can be made consistent by removing blue from NSW

- **Continue to propagate constraints:**

    – Check V $\rightarrow$ NSW : not consistent for V = red; remove red from V

Arc consistency

- Simplest form of propagation makes each arc consistent

- $X \rightarrow Y$ is consistent iff

    for every value x of X there is some allowed value y for Y (note: directed!)
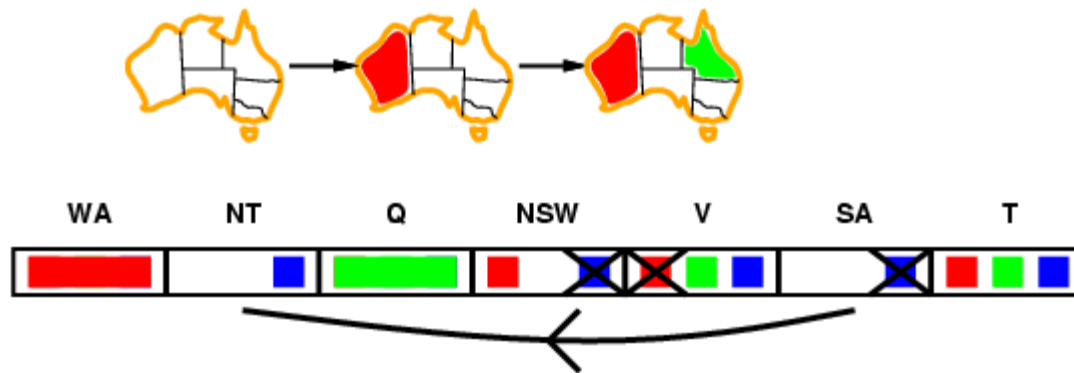

- Continue to propagate constraints

- SA $\rightarrow$ NT not consistent:

    - **And cannot be made consistent! Failure!**

- Arc consistency detects failure earlier than FC

    – But requires more computation: is it worth the effort?

Local search: min-conflicts heuristic

- Use complete-state representation
  - Initial state = all variables assigned values
  - Successor states = change 1 (or more) values

- For CSPs
  - allow states with unsatisfied constraints (unlike backtracking)
  - operators **reassign** variable values
  - hill-climbing with n-queens is an example

- **Variable selection:** randomly select any conflicted variable
- **Value selection:** *min-conflicts heuristic*
  - Select new value that results in a minimum number of conflicts with the other variables

Local search: min-conflicts heuristic

function MIN-CONFLICTS(*csp, max_steps*) **return** solution or failure
    inputs: *csp*, a constraint satisfaction problem
        max_steps, the number of steps allowed before giving up

    current ←   a (random) initial complete assignment for *csp*
    for *i* = 1 to *max_steps* **do**
        if *current* is a solution for *csp* then return *current*
        var ← a randomly chosen, conflicted variable from
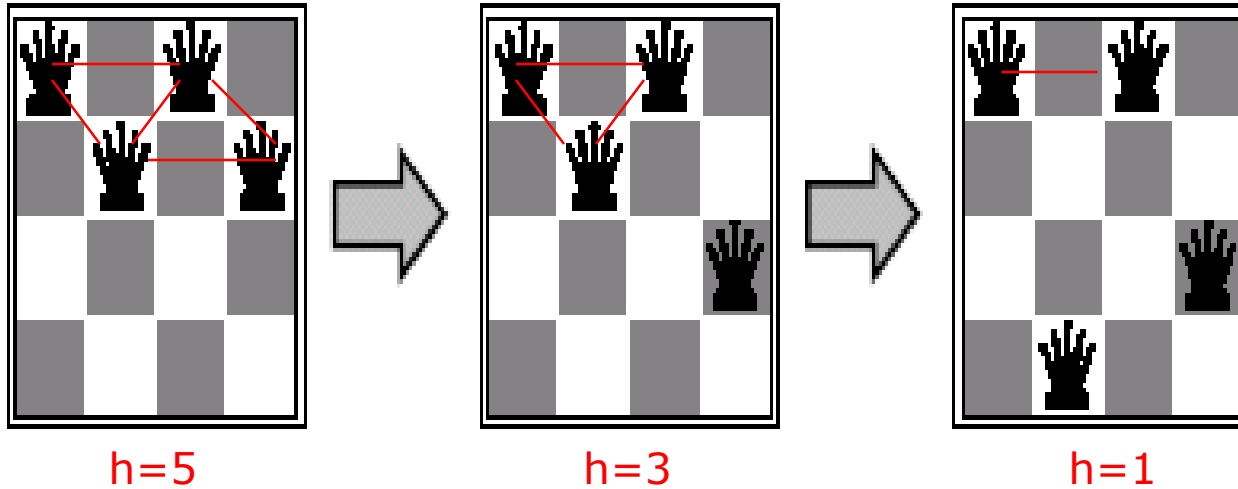   VARIABLES[*csp*]
        value ← the value *v* for *var* that minimize
CONFLICTS(*var,v,current,csp*)
        set *var = value* in *current*
    return *failure*

Min-conflicts example 1

h=5   h=3   h=1

Use of min-conflicts heuristic in hill-climbing.

Summary

- CSPs
  - special kind of problem: states defined by values of a fixed set of variables, goal test defined by constraints on variable values

- Backtracking = depth-first search, one variable assigned per node

- Heuristics: variable order & value selection heuristics help a lot

- Constraint propagation
  - does additional work to constrain values and detect inconsistencies
  - Works effectively when combined with heuristics

- Iterative min-conflicts is often effective in practice.

- Graph structure of CSPs determines problem complexity
  - e.g., tree structured CSPs can be solved in linear time.

Review Propositional Logic
Chapter 7.1-7.5; Optional 7.6-7.8

- Definitions:
  - Syntax, Semantics, Sentences, Propositions, Entails, Follows, Derives, Inference, Sound, Complete, Model, Satisfiable, Valid (or Tautology)

- Syntactic & Semantic Transformations:
  - E.g., $(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$
  - E.g., $(KB \models \alpha) \equiv (\models (KB \Rightarrow \alpha)$

- Truth Tables:
  - Negation, Conjunction, Disjunction, Implication, Equivalence (Biconditional)

- Inference:
  - By Resolution (CNF)
  - By Backward & Forward Chaining (Horn Clauses)
  - By Model Enumeration (Truth Tables)

Recap propositional logic: Syntax

- Propositional logic is the simplest logic – illustrates basic ideas

- The proposition symbols P_1, P_2 etc are sentences

  - If S is a sentence, $\neg$S is a sentence (negation)
  - If S_1 and S_2 are sentences, $S_1 \wedge S_2$ is a sentence (conjunction)
  - If S_1 and S_2 are sentences, $S_1 \vee S_2$ is a sentence (disjunction)
  - If S_1 and S_2 are sentences, $S_1 \Rightarrow S_2$ is a sentence (implication)
  - If S_1 and S_2 are sentences, $S_1 \Leftrightarrow S_2$ is a sentence (biconditional)

Recap propositional logic:
Semantics

Each model/world specifies true or false for each proposition symbol

    E.g., $P_{1,2}$ $P_{2,2}$ $P_{3,1}$
                           false   true   false

    With these symbols, 8 possible models can be enumerated automatically.

Rules for evaluating truth with respect to a model *m*:

      $\neg S$ is true iff   S is false
      $S_1 \wedge S_2$ is true iff S_1 is true and S_2 is true
      $S_1 \vee S_2$ is true iff S_1 is true or S_2 is true
      $S_1 \Rightarrow S_2$ is true iff S_1 is false or S_2 is true
      (i.e.,   is false iff S_1 is true and S_2 is false
      $S_1 \Leftrightarrow S_2$ is true iff $S_1 \Rightarrow S_2$ is true and $S_2 \Rightarrow S_1$ is true

Simple recursive process evaluates an arbitrary sentence, e.g.,

         $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$ = *true* $\wedge$ (*true* $\vee$ *false*) = *true* $\wedge$ *true* = *true*

Recap propositional logic:
Truth tables for connectives

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

OR: P or Q is true or both are true.
XOR: P or Q is true but not both.

Implication is always true when the premises are False!

Recap propositional logic:
Logical equivalence and rewrite rules

- To manipulate logical sentences we need some rewrite rules.
- Two sentences are logically equivalent iff they are true in same models: $\alpha \equiv \beta$ iff $\alpha \models \beta$ and $\beta \models \alpha$

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$
$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$
$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$
$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{de Morgan}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{de Morgan}$$
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$

You need to know these !

Entailment

- Entailment means that one thing follows from another set of things:

$$\text{KB} \models \alpha$$

- Knowledge base *KB* entails sentence α if and only if α is true in all worlds wherein *KB* is true

  - E.g., the KB = "the Giants won and the Reds won" entails α = "The Giants won".
  - E.g., KB = "x+y = 4" entails α = "4 = x+y"
  - E.g., KB = "Mary is Sue's sister and Amy is Sue's daughter" entails α = "Mary is Amy's aunt."

- The entailed α <u>MUST BE TRUE</u> in <u>ANY</u> world in which <u>KB IS TRUE</u>.

Review: Models (and in FOL, Interpretations)

- Models are formal worlds in which truth can be evaluated

- We say m is a model of a sentence α if α is true in m

- $M(\alpha)$ is the set of all models of α

- Then KB $\models$ α iff $M(KB) \subseteq M(\alpha)$
  - E.g. *KB,* = "Mary is Sue's sister and Amy is Sue's daughter."
  - α = "Mary is Amy's aunt."

- Think of KB and α as constraints, and of models m as possible states.
- M(KB) are the solutions to KB and M(α) the solutions to α.
- Then, KB $\models$ α, i.e., $\models$ (KB $\Rightarrow$ a) , when all solutions to KB are also solutions to α.

M(◯)

M(KB)
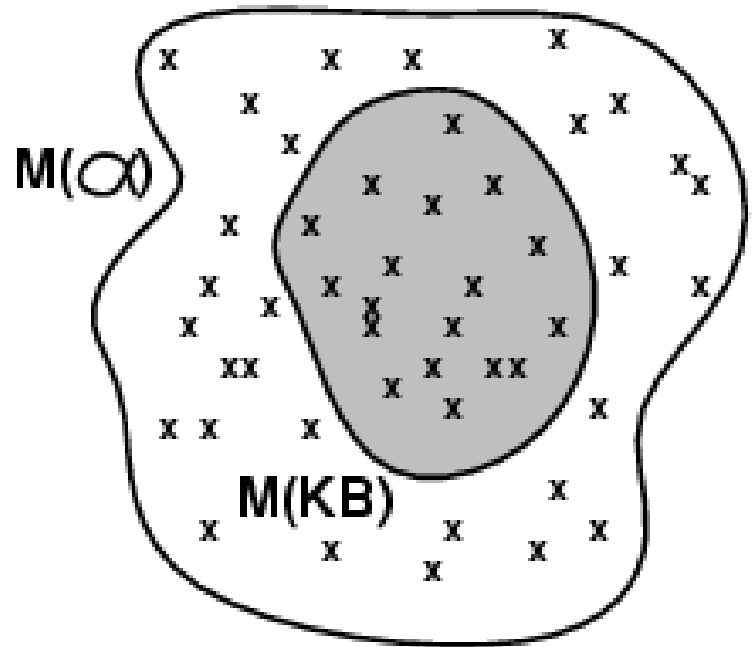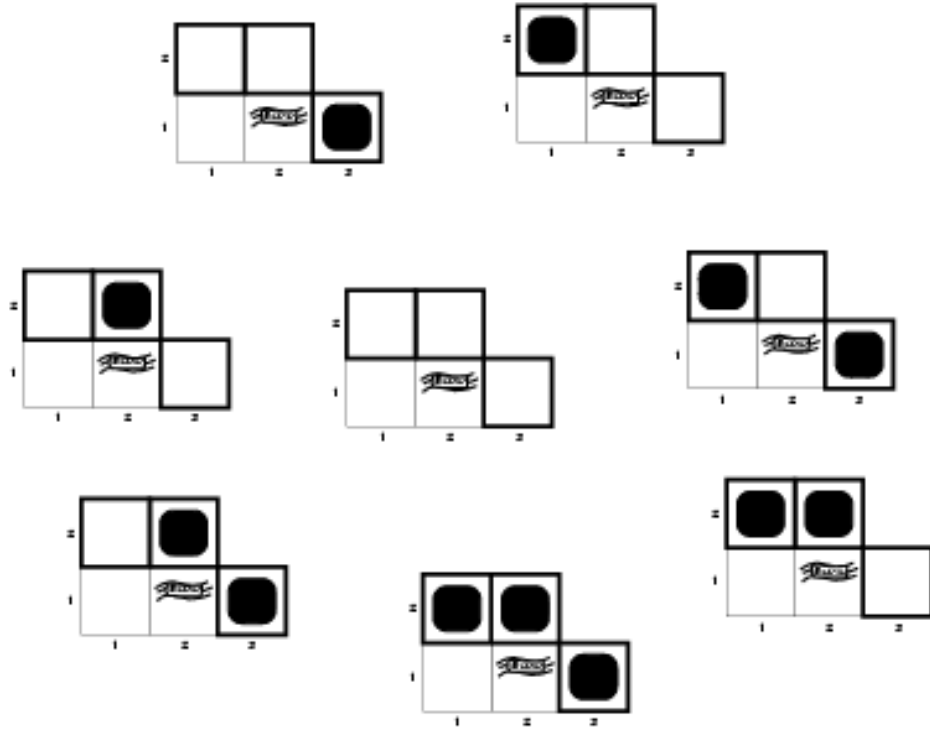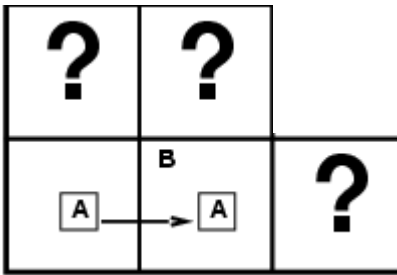
Wumpus models

All possible models in this reduced Wumpus world. What can we infer?

Review: Wumpus models
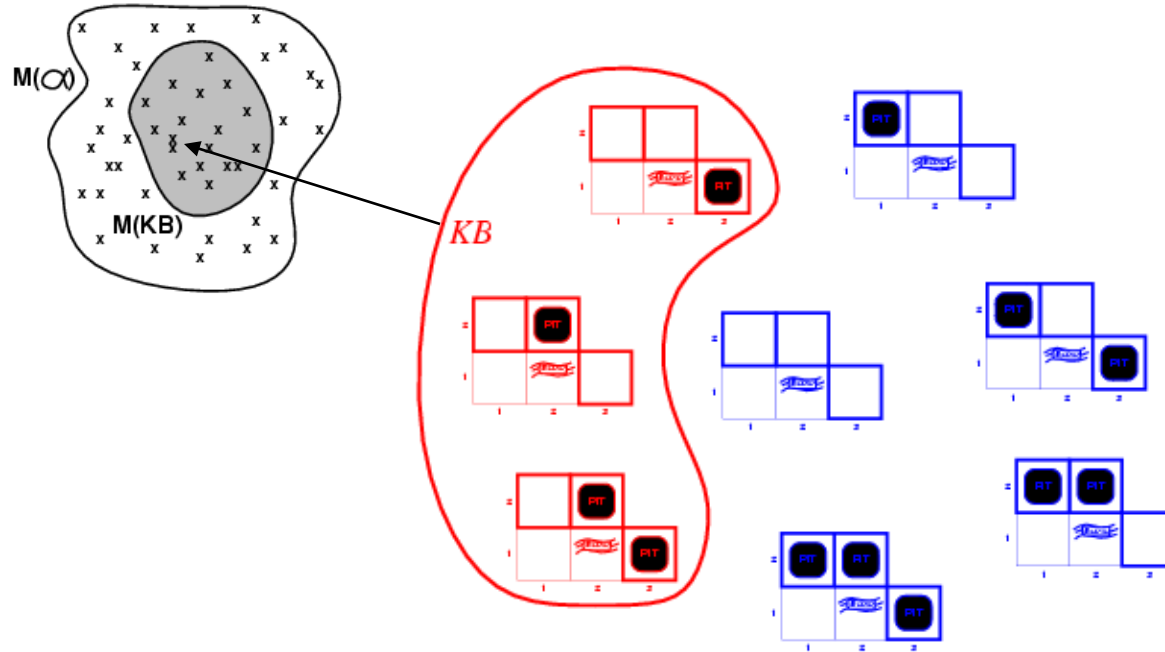
- *KB* = all possible wumpus-worlds consistent with the observations and the "physics" of the Wumpus world.

Wumpus models
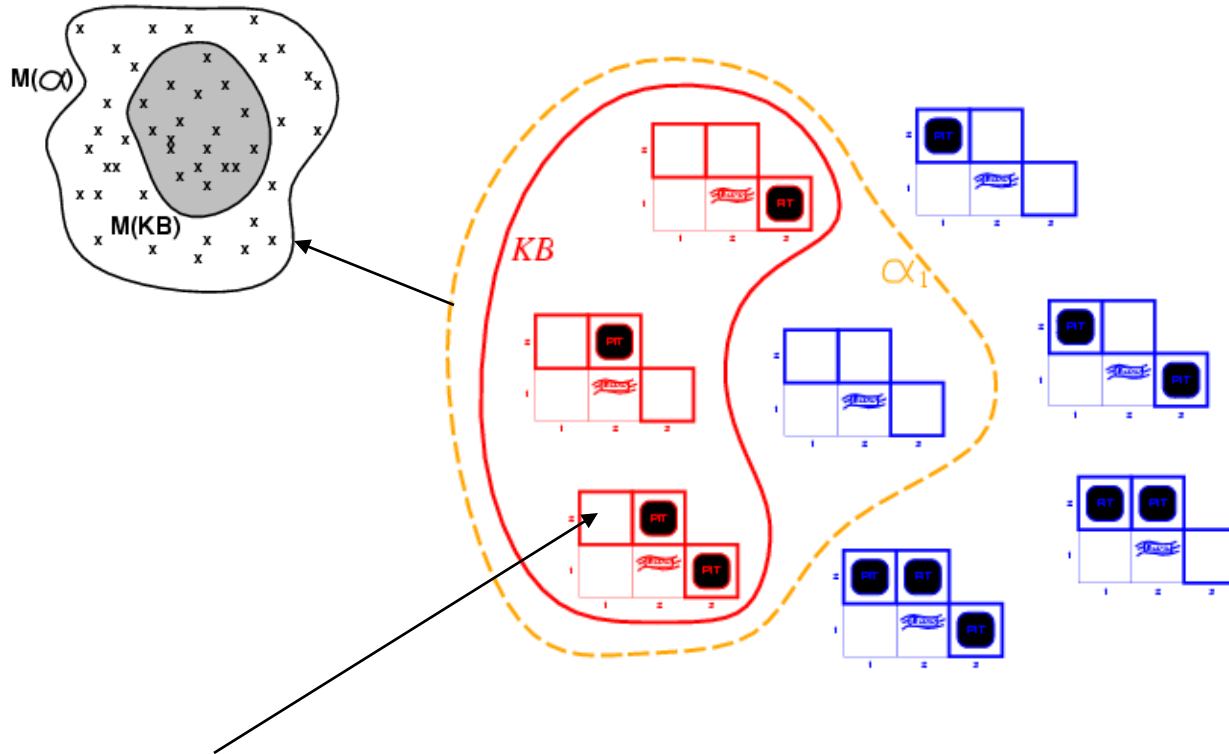
Now we have a query sentence, α_1 = "[1,2] is safe"

$KB \models \alpha_1$, proved by **model checking**

M(KB) (red outline) is a subset of M(α_1) (orange dashed outline)

$\Rightarrow$ **α_1 is true in any world in which KB is true**

Wumpus models

Now we have another query sentence, α_2 = "[2,2] is safe"

$KB \not\models \alpha_2$, proved by **model checking**

M(KB) (red outline) is a **not** a subset of M(α_2) (dashed outline)

$\Rightarrow$ **α_2 is false in some world(s) in which KB is true**

Recap propositional logic: Validity and satisfiability

A sentence is valid if it is true in all models,

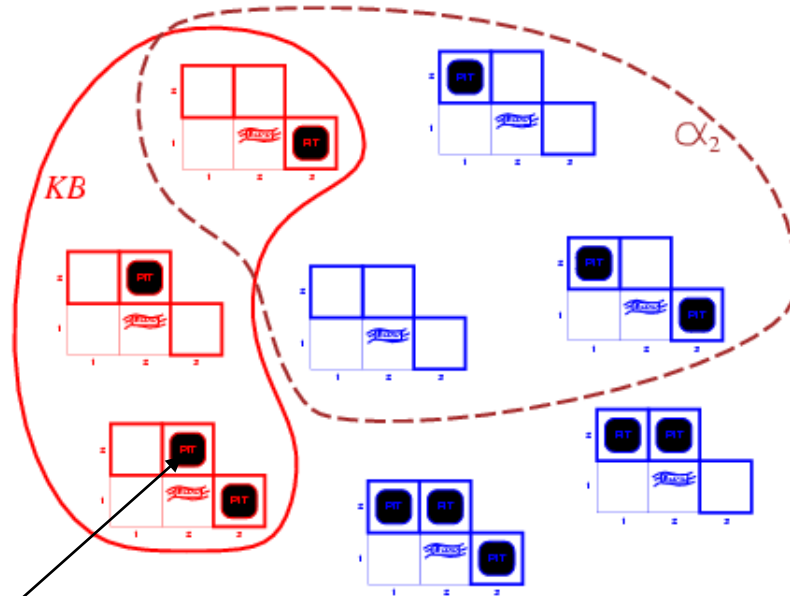e.g., *True*, $\quad A \vee \neg A, \quad A \Rightarrow A, \quad (A \wedge (A \Rightarrow B)) \Rightarrow B$

Validity is connected to inference via the Deduction Theorem:

$KB \models \alpha$ if and only if $(KB \Rightarrow \alpha)$ is valid

A sentence is satisfiable if it is true in some model

e.g., $A \vee B, \quad C$

A sentence is unsatisfiable if it is false in all models

e.g., $A \wedge \neg A$

Satisfiability is connected to inference via the following:

$KB \models A$ if and only if $(KB \wedge \neg A)$ is unsatisfiable
(there is no model for which KB is true and A is false)

Logical inference

- The notion of entailment can be used for logic inference.
  - Model checking (see wumpus example):
    enumerate all possible models and check whether α is true.

- *KB $|$-$_i$ α* means *KB* derives a sentence α using inference procedure *i*

- *Sound* (or *truth preserving*):
  The algorithm **only** derives entailed sentences.
  - Otherwise it just makes things up.
    i is sound iff whenever KB $|$-$_i$ α it is also true that KB$|$= α
  - *E.g., model-checking is sound*
  Refusing to infer any sentence is Sound; so, Sound is weak alone.
- *Complete*:
  The algorithm can derive **every** entailed sentence.
    i is complete iff whenever KB $|$= α it is also true that KB$|$-$_i$ α
  Deriving every sentence is Complete; so, Complete is weak alone.

Inference by Resolution

- KB is represented in CNF
  - KB = AND of all the sentences in KB
  - KB sentence = clause = OR of literals
  - Literal = propositional symbol or its negation

- Find two clauses in KB, one of which contains a literal and the other its negation
  - Cancel the literal and its negation
  - Bundle everything else into a new clause
  - Add the new clause to KB
  - Repeat

Example: Conversion to CNF

Example: $B_{1,1} \Leftrightarrow (P_{1,2} \lor P_{2,1})$

1. Eliminate $\Leftrightarrow$ by replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)$.
   $= (B_{1,1} \Rightarrow (P_{1,2} \lor P_{2,1})) \land ((P_{1,2} \lor P_{2,1}) \Rightarrow B_{1,1})$

2. Eliminate $\Rightarrow$ by replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \lor \beta$ and simplify.
   $= (\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land (\neg(P_{1,2} \lor P_{2,1}) \lor B_{1,1})$

3. Move $\neg$ inwards using de Morgan's rules and simplify.
   $$\neg(\alpha \lor \beta) \equiv (\neg\alpha \land \neg\beta), \; \neg(\alpha \land \beta) \equiv (\neg\alpha \lor \neg\beta)$$
   $= (\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land ((\neg P_{1,2} \land \neg P_{2,1}) \lor B_{1,1})$

4. Apply distributive law ($\land$ over $\lor$) and simplify.
   $= (\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land (\neg P_{1,2} \lor B_{1,1}) \land (\neg P_{2,1} \lor B_{1,1})$

Example: Conversion to CNF

Example: $B_{1,1} \Leftrightarrow (P_{1,2} \lor P_{2,1})$

From the previous slide we had:

$= (\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land (\neg P_{1,2} \lor B_{1,1}) \land (\neg P_{2,1} \lor B_{1,1})$

5. KB is the conjunction of all of its sentences (all are true), so write each clause (disjunct) as a sentence in KB:

KB =   **Often, Won't Write "$\lor$" or "$\land$"**
                                    (we know they are there)
...

$(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1})$ $\longrightarrow$ $(\neg B_{1,1} \quad P_{1,2} \quad P_{2,1})$
$(\neg P_{1,2} \lor B_{1,1})$ $\longrightarrow$ $(\neg P_{1,2} \quad B_{1,1})$
$(\neg P_{2,1} \lor B_{1,1})$ $\longrightarrow$ $(\neg P_{2,1} \quad B_{1,1})$

...   **(same)**

Resolution = Efficient Implication

Recall that (A => B) = ((NOT A) OR B)
and so:
$\qquad$ (Y OR X) = ((NOT X) => Y)
((NOT Y) OR Z) = (Y => Z)
which yields:
$\qquad$ ((Y OR X) AND ((NOT Y) OR Z)) = ((NOT X) => Z) = (X OR Z)

(OR A B C D)   ->Same ->   (NOT (OR B C D)) => A
(OR ¬A E F G)   ->Same ->   A => (OR E F G)
-----------------------------   ---
(OR B C D E F G)   (NOT (OR B C D)) => (OR E F G)

                                        (OR B C D E F G)

Recall: All clauses in KB are conjoined by an implicit AND (= CNF representation).

Resolution Examples

- Resolution: inference rule for CNF: sound and complete! *

$(A \lor B \lor C)$

$(\neg A)$ "If A or B or C is true, but not A, then B or C must be true."

— — — — — — — — — — — —

$\therefore (B \lor C)$

$(A \lor B \lor C)$

$(\neg A \lor D \lor E)$ "If A is false then B or C must be true, or if A is true then D or E must be true, hence since A is either true or false, B or C or D or E must be true."

— — — — — — — — — — — —

$\therefore (B \lor C \lor D \lor E)$

$(A \lor B)$ "If A or B is true, and not A or B is true, then B must be true."

$(\neg A \lor B)$

— — — — — — — — —

$\therefore (B \lor B) \equiv B$ ⟵ Simplification is done always.

> * Resolution is "refutation complete" in that it can prove the truth of any entailed sentence by refutation.

More Resolution Examples

1. (P Q ¬R S) with (P ¬Q W X) yields <u>(P ¬R S W X)</u>
   <u>Order of literals within clauses does not matter.</u>
2. (P Q ¬R S) with (¬P) yields <u>(Q ¬R S)</u>
3. (¬R) with (R) yields <u>( ) or FALSE</u>
4. (P Q ¬R S) with (P R ¬S W X) yields <u>(P Q ¬R R W X) or (P Q S ¬S W X) or TRUE</u>
5. (P ¬Q R ¬S) with (P ¬Q R ¬S) yields <u>None possible (no complementary literals)</u>
6. (P ¬Q ¬S W) with (P R ¬S X) yields <u>None possible (no complementary literals)</u>
7. ((¬ A) (¬ B) (¬ C) (¬ D)) with ((¬ C) D) yields <u>((¬ A) (¬ B) (¬ C))</u>
8. ((¬ A) (¬ B) (¬ C)) with ((¬ A) C) yields <u>((¬ A) (¬ B))</u>
9. ((¬ A) (¬ B)) with (B) yields (¬ A)
10. (A C) with (A (¬ C)) yields <u>(A)</u>
11. (¬ A) with (A) yields <u>( ) or FALSE</u>

Only Resolve <u>ONE</u> Literal Pair!
If more than one pair, result always = TRUE.
<u>Useless!!</u> Always simplifies to TRUE!!

No!
(OR   A   B   C   D)
(OR ¬A ¬B   F   G)

(OR C D F G)
No! This is wrong!

Yes! (but = TRUE)
(OR   A   B   C   D)
(OR ¬A ¬B   F   G)

(OR B ¬B C D F G)
Yes! (but = TRUE)

No!
(OR   A   B   C   D)
(OR ¬A ¬B ¬C)

(OR D)
No! This is wrong!

Yes! (but = TRUE)
(OR   A   B   C   D)
(OR ¬A ¬B ¬C)

(OR A ¬A B ¬B D)
Yes! (but = TRUE)

Resolution Algorithm

- The resolution algorithm tries to prove: $KB \models \alpha$ *equivalent to*

  $KB \wedge \neg\alpha$ *unsatisfiable*

- Generate all new sentences from KB and the (negated) query.
- One of two things can happen:

1. We find $P \wedge \neg P$ which is unsatisfiable. I.e. we can entail the query.

2. We find no contradiction: there is a model that satisfies the sentence $KB \wedge \neg\alpha$ (non-trivial) and hence we cannot entail the query.

Resolution example
Resulting Knowledge Base stated in CNF

- "Laws of Physics" in the Wumpus World:
  $(\neg B_{1,1} \quad P_{1,2} \quad P_{2,1})$
  $(\neg P_{1,2} \quad B_{1,1})$
  $(\neg P_{2,1} \quad B_{1,1})$

- Particular facts about a specific instance:

  $(\neg B_{1,1})$

- <u>Negated</u> goal or query sentence:

  $(P_{1,2})$

Resolution example
A Resolution proof ending in ( )

- # Knowledge Base at start of proof:

$$(\neg B_{1,1} \quad P_{1,2} \quad P_{2,1})$$
$$(\neg P_{1,2} \quad B_{1,1})$$
$$(\neg P_{2,1} \quad B_{1,1})$$
$$(\neg B_{1,1})$$
$$(P_{1,2})$$

A resolution proof ending in ( ):

- Resolve $(\neg P_{1,2} \quad B_{1,1})$ and $(\neg B_{1,1})$ to give $(\neg P_{1,2})$

- Resolve $(\neg P_{1,2})$ and $(P_{1,2})$ to give ( )

- Consequently, the goal or query sentence is entailed by KB.
- Of course, there are many other proofs, which are OK iff correct.

Detailed Resolution Proof Example

- **In words:** *If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.*

  Prove that the unicorn is both magical and horned.

  ((NOT Y) (NOT R))   (M Y)   (R Y)   (H (NOT M))

  (H R)   ((NOT H) G)   ((NOT G) (NOT H))

- **Fourth, produce a resolution proof ending in ( ):**

- Resolve (¬H ¬G) and (¬H G) to give (¬H)

- Resolve (¬Y ¬R) and (Y M) to give (¬R M)

- Resolve (¬R M) and (R H) to give (M H)

- Resolve (M H) and (¬M H) to give (H)

- Resolve (¬H) and (H) to give ( )

- Of course, there are many other proofs, which are OK iff correct.

Horn Clauses

- Resolution can be exponential in space and time.

- If we can reduce all clauses to "Horn clauses" inference is linear in space and time

A clause with at most 1 positive literal.

e.g. $A \vee \neg B \vee \neg C$

- Every Horn clause can be rewritten as an implication with a conjunction of positive literals in the premises and at most a single positive literal as a conclusion.

  e.g. $A \vee \neg B \vee \neg C \equiv B \wedge C \Rightarrow A$

- 1 positive literal and $\geq$ 1 negative literal: definite clause (e.g., above)
- 0 positive literals: integrity constraint or goal clause

  e.g. $(\neg A \vee \neg B) \equiv (A \wedge B \Rightarrow False)$ states that $(A \wedge B)$ must be false

- 0 negative literals: fact

  e.g., $(A) \equiv (True \Rightarrow A)$ states that A must be true.

- Forward Chaining and Backward chaining are sound and complete with Horn clauses and run linear in space and time.

Propositional Logic --- Summary

- Logical agents apply inference to a knowledge base to derive new information and make decisions

- Basic concepts of logic:
    - syntax: formal structure of sentences
    - semantics: truth of sentences wrt models
    - entailment: necessary truth of one sentence given another
    - inference: deriving sentences from other sentences
    - soundness: derivations produce only entailed sentences
    - completeness: derivations can produce all entailed sentences
    - valid: sentence is true in every model (a tautology)

- Logical equivalences allow syntactic manipulations

- Propositional logic lacks expressive power
    - Can only state specific facts about the world.
    - Cannot express general rules about the world
      (use First Order Predicate Logic instead)

Midterm Review

- Agents: R&N Chap 2.1-2.3

- State Space Search: R&N Chap 3.1-3.7

- Local Search: R&N Chap 4.1-4.2

- Adversarial (Game) Search: R&N Chap 5.1-5.4

- Constraint Satisfaction: R&N Chap 6.1-6.4 (except 6.3.3)

- Propositional Logic: R&N Chap 7.1-7.5